

AD-A183 782

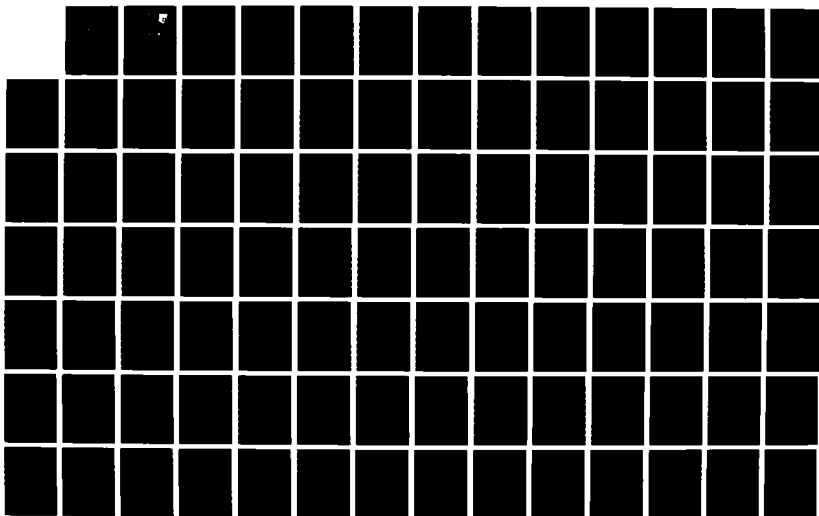
MULTILAYER NETWORKS OF SELF-INTERESTED ADAPTIVE UNITS
(U) MASSACHUSETTS UNIV AMHERST DEPT OF COMPUTER AND
INFORMATION SCIENCE A G BARTO JUL 87 AFMIL-TR-87-1052
F33615-83-C-1078

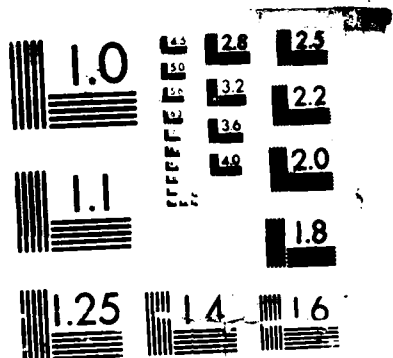
1/2

UNCLASSIFIED

F/G 12/7

NL





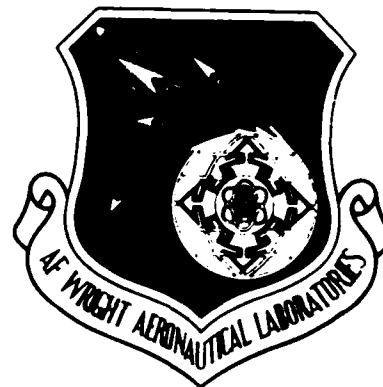
MICROCOPY RESOLUTION TEST CHART

DTIC FILE COPY

2

AD-A183 782

AFWAL-TR-87-1052



MULTILAYER NETWORKS OF SELF-INTERESTED
ADAPTIVE UNITS

Andrew G. Barto, Editor
Computer and Information Science
University of Massachusetts
Amherst, MA 01003

DTIC
ELECTE
AUG 26 1987
S D

July 1987

FINAL REPORT for period September 1983 - September 1986

Approved for public release; distribution is unlimited

AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

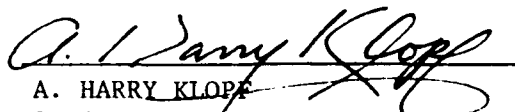
87 8 18 057

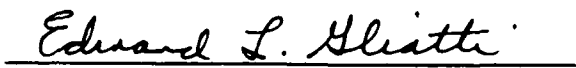
NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

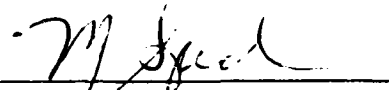
This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.


A. HARRY KLOP
Project Engineer
Advanced Systems Research Group
Avionics Laboratory


EDWARD L. GLIATTI, Chief
Information Processing Technology Br
Avionics Laboratory

FOR THE COMMANDER


MARVIN SPECTOR, Chief
System Avionics Division
Avionics Laboratory

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT-3, Wright-Patterson AFB, OH 45433- 6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

A155782

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			5. MONITORING ORGANIZATION REPORT NUMBER(S) AFWAL-TR-87-1052		
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			6a. NAME OF PERFORMING ORGANIZATION University of Massachusetts Computer and Information Science Department		
6b. OFFICE SYMBOL (if applicable)			7a. NAME OF MONITORING ORGANIZATION Air Force Wright Aeronautical Laboratories Avionics Laboratory (AFWAL/AAAT-3)		
6c. ADDRESS (City, State, and ZIP Code) Amherst, MA 01003			7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6543		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR		8b. OFFICE SYMBOL (if applicable) NL		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33615-83-C-1078	
8c. ADDRESS (City, State, and ZIP Code) Bolling AFB, DC 20332		10. SOURCE OF FUNDING NUMBERS			
		PROGRAM ELEMENT NO. 61102F		PROJECT NO. 2312	
		TASK NO. R1		WORK UNIT ACCESSION NO. 04	
11. TITLE (Include Security Classification) Multilayer Networks of Self-Interested Adaptive Units					
12. PERSONAL AUTHOR(S) Andrew G. Barto, Editor					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM 09-83 TO 09-86		14. DATE OF REPORT (Year, Month, Day) July 1987	
15. PAGE COUNT 149					
16. SUPPLEMENTARY NOTATION F-TR-87					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
			Adaptive Networks Learning		
			Neural Computing Stochastic Learning Automata		
			Connectionist Systems Cooperative Computing		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This report describes research directed toward refining and evaluating learning methods for multilayer networks of neuron-like adaptive units. We define a learning rule called the Associative Reward-Penalty, or Ar-p, rule that has strong ties to both the theory of adaptive pattern classification and stochastic learning automata. We state a convergence result that has been proven for a single Ar-p unit and show, via computer simulation, how layered networks of Ar-p units can reliably learn nonlinear associative mappings. The behavior of these networks is discussed in terms of the collective behavior of stochastic learning automata in team decision problems. A number of methods for learning in multilayer networks are compared, including the Ar-p method and the error back-propagation method. These methods, or variants of them, outperform the other methods applied to the test problem, with error back-propagation showing a significant speed advantage over the other					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL A. Harry Klopff			22b. TELEPHONE (Include Area Code) 513-255-7649		22c. OFFICE SYMBOL AFWAL/AAAT-3

UNCLASSIFIED

19. Abstract (Continued)

A Sub R-P

methods. The ~~Ar-p~~ and error back-propagation are compared and contrasted in terms of their respective approaches to gradient following.

Simulation experiments are described in which layered adaptive networks are applied to learning strategies for a pole-balancing task and a Tower of Hanoi puzzle. These results extend earlier results by replacing the fixed encoding of network input by an adaptive encoding mechanism implemented by a layered network. Comparisons are made between the performance of one- and two-layer networks on these tasks, and the results are related to more conventional approaches to control in engineering and problem solving in artificial intelligence.

Accession For	
NTIS CRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



UNCLASSIFIED

ACKNOWLEDGEMENTS

First, I would like to thank Charles W. Anderson for permitting me to include parts of his dissertation in this report. Sections 4, 5, and 6 are condensed versions of Chapters IV, V, VI, and VII of his dissertation [4], and Section 7 contains some material from Chapter VIII. Although I rearranged and edited this material, its authorship remains with Chuck. It is mainly for this reason that I am listed as the editor, not the author, of this report. Of course, not only did Chuck write this material, he also conducted the research it describes: he designed the simulation experiments—selected or devised the methods included in the comparative study described in Section 4, implemented the methods, and interpreted the results. Beginning where we left off with the pole-balancing system as reported in our 1983 paper [13], Chuck designed, implemented, and tested the extensions reported here. Another reason that I am editor of this report and not author is that parts of Section 2 are abstracted from a paper I coauthored with P. Anandan which appeared in *IEEE Transactions on Systems, Man, and Cybernetics* [7]. Anandan deserves much of the credit for formulating the A_{R-P} learning rule and nearly all of the credit for proving the convergence theorem. Section 3 contains material abstracted from my paper in *Human Neurobiology* [6].

Others contributed to the research described here in a variety of indispensable ways. Harry Klopf provided us with the provocative idea of self-interested network components, which we continue to find thought-provoking and fruitful. Rich Sutton's experiments with reinforcement-learning methods contributed to the foundations for the present work, and the temporal credit-assignment method he has developed and refined forms an important part of the strategy-learning networks described here. John Moore shared with us his great knowledge of animal learning behavior and mechanisms, not to mention his enthusiasm and wit. Michael Arbib helped us appreciate the history of this field and created an environment in which we could study

it - we miss him since he went to the University of Southern California. Michael Jordan joined us near the end of the period covered by this report as a post-doctoral researcher. He brought a lot of new ideas that will play prominent roles in future reports. Jonathan Bachrach, Robbie Jacobs, Stephen Judd, Brian Pinette, and Mike Seymour have all begun promising research that should also figure prominently in future reports. I thank Neil Berthier, Ross Beveridge, Diana Blazis, David Day, John Desmond, and Ira Smotroff, who have pursued topics related to connectionist research, and all the others who have contributed to the high level of interest in connectionist research at the University of Massachusetts. I thank Susan Parker who applied her considerable administrative skills to the project, untangling a lot of red tape in the process. Finally, I wish to thank the connectionists out there for making this field receive all of the attention it deserves—and more.

TABLE OF CONTENTS

SECTION	PAGE
1. INTRODUCTION	1
2. THE ASSOCIATIVE REWARD-PENALTY UNIT	5
The Associative Reinforcement Learning Task	7
The A_{R-P} Learning Rule	7
Simulation of a Single A_{R-P} Unit	10
A_{R-P} Units and Stochastic Learning Automata	12
3. COOPERATIVE BEHAVIOR OF A_{R-P} UNITS	18
Associative Search Networks and Team Decision Problems	18
Layered Teams of A_{R-P} Units	20
A Minimal Layered Network of A_{R-P} Units	21
The XOR Task	25
The Multiplexer Task	27
Discussion— A_{R-P} Networks and Gradient Descent	31
4. COMPARATIVE STUDIES OF LAYERED NETWORK LEARNING METHODS	34
Direct-Search Methods	36
Unguided Random Search	36
Guided Random Search	38
The Polytope Algorithm	39
Error Back-Propagation Methods	41
Rosenblatt's Back-Propagation Method	41
Rumelhart, Hinton, and Williams	42
Associative Reinforcement Learning	47
Associative Search with Reinforcement Prediction	48
Associative Reward-Penalty	51
Local Reinforcement	52
Penalty Prediction	55
Summary of Comparative Simulations	59
Some Further Experiments—The Batched A_{R-P} Method	66

TABLE OF CONTENTS (Concluded)

SECTION	PAGE
5. POLE-BALANCING AGAIN	70
Strategy Learning Networks	71
Output Functions	73
Learning Rules	75
The Pole-Balancing Task	80
Interaction between the Network and Cart-Pole Simulation	85
Results	86
One-Layer Experiments	86
Results of Two-Layer Experiments	91
Conclusion	99
6. LEARNING TO SOLVE A PUZZLE	100
The Tower of Hanoi Puzzle	100
Representation of States and Actions	103
Reinforcement	105
Results of One-Layer Experiments	107
Results of Two-Layer Experiments	110
Transfer of Learning	117
Conclusion	119
7. SUMMARY AND CONCLUSIONS	120
The Associative Reward-Penalty Unit	120
Cooperative Behavior of A_{R-P} Units	121
Comparison of Methods for Learning by Layered Networks	123
Strategy Learning with Multilayer Networks	125
Pole Balancing	126
Tower of Hanoi	127
Further Developments of Strategy Learning Networks	129
BIBLIOGRAPHY	132

LIST OF ILLUSTRATIONS

FIGURE	PAGE
1. Simulation Results for a Single A_{R-P} Unit	13
2. Stochastic Learning Automaton Interacting with a Random Environment.	14
3. (a) The game problem. (b) The team problem.	15
4. Associative search network.	19
5. A Minimal Layered Network of A_{R-P} Units	21
6. Simulation Results for the Two-unit Network.	24
7. Network for the Exclusive-Or Task.	26
8. Behavior of Network Units in a Typical Sequence of 5000 Trials in the Exclusive-Or Task.	28
9. Histogram of Trials to Criterion for 100 Sequences of Trials in the Exclusive-Or Task.	29
10. Layered Network for the Multiplexer Problem.	29
11. Histogram of Trials to Criterion for the Multiplexer Task.	30
12. Learning Curves for Error Back-propagation Methods	43
13. Learning Curve for Reinforcement Learning Methods on the Multiplexer Task	50
14. Learning Curves for All Methods on the Multiplexer Task	60
15. Learning Curves Comparing the Standard and Batched A_{R-P} Methods	68
16. Two-Layer Networks for Strategy Learning	72
17. Good Functions for Pole-Balancing Solution	82
18. Balancing Time versus Trials for One-Layer System	88
19. Weights Learned by One-Layer Network	89
20. Functions Learned by One-Layer Networks	90
21. Balancing Time versus Trials for Two-Layer System	92
22. Weights Learned by Two-Layer Network	94
23. Functions Learned by Two-Layer Networks	95
24. New Features Learned by Two-Layer Networks	98
25. Initial and Goal States of the Tower of Hanoi Puzzle	101
26. State Transition Graph for Three-Disk Tower of Hanoi Puzzle	102

LIST OF ILLUSTRATIONS (Concluded)

FIGURE	PAGE
27. Length of Solution Path versus Trials for One-Layer System	108
28. Weights Learned by One-Layer Network	109
29. Evaluation Function Learned by One-Layer Network	110
30. Length of Solution Path versus Trials for Two-Layer System	112
31. Weights Learned by Two-Layer Network	113
32. Evaluation Function Learned by Two-Layer Network	113
33. New Features Learned by Two-Layer Evaluation Network	114

LIST OF TABLES

TABLE	PAGE
1. Unguided Random Search on the Multiplexer Task	37
2. Guided Random Search on the Multiplexer Task	39
3. Polytope Algorithm on the Multiplexer Task	40
4. Rosenblatt's Back-Propagation Method on the Multiplexer Task	43
5. Rumelhart et al. Error Back-propagation Method on the Multiplexer Task	45
6. Sutton's Modification of the Error Back-propagation Method on the Multiplexer Task	47
7. Associative Search with Reinforcement Prediction on the Multiplexer Task	50
8. A_{R-P} Method on the Multiplexer Task	52
9. A_{R-P} with Local Reinforcement on the Multiplexer Task	54
10. A_{R-P} with Penalty Prediction on the Multiplexer Task	58
11. Performance Summary for Multiplexer Task	61
12. New Features Developed by the Error Back-Propagation Method	63
13. New Features Developed by A_{R-P} with Penalty Prediction Method	65
14. Results of One-Layer System	87
15. Results of Two-Layer System	92
16. Results of One-Layer System	107
17. Results of Two-Layer System	111

SECTION 1

INTRODUCTION

This report describes progress made by our research group over the period September 1983 to September 1986. Although we explored a range of issues in connectionist learning, the major focus was the study of learning nonlinear associative mappings by layered networks. In earlier research we obtained some preliminary results with an approach to this problem based on reinforcement learning [10,3,15]. However, the specific reinforcement learning rules used in these studies did not produce rapid and reliable learning in all the learning tasks we tried. During the period reported on here, we set out to obtain better understanding of this class of methods through computer simulation and mathematical analysis.

The research direction that proved most fruitful was our effort to develop rigorous ties between our reinforcement-learning adaptive units and the theory of stochastic learning automata. Our initial aim was to develop a theoretically tractable learning rule by developing one that specialized, under one set of restrictions, to a familiar supervised-learning rule while also specializing, under another set of restrictions, to one of the simplest of the stochastic learning automaton algorithms. The result is a learning rule that we call the *Associative Reward-Penalty*, or A_{R-P} , learning rule. It is very closely related to a relatively little-known learning rule presented by Widrow, Gupta, and Maitra [58] that they called the "selective bootstrap adaptation" rule. Thus, although it is novel, the A_{R-P} rule is closely connected to existing theory, and we were able to prove a convergence theorem for a single adaptive unit implementing the A_{R-P} rule (we call such a unit an A_{R-P} unit) [7]. What was rather surprising was that the A_{R-P} unit turned out to perform very well as a network component. Layered networks of A_{R-P} units solve nonlinear associative learning problems with great reliability. This surprised us because in devising the A_{R-P} unit we were concerned

solely with the mathematical tractability of a single unit and not with network performance. However, it later became apparent that the learning capabilities that single A_{R-P} units provably possess are crucial in obtaining reliable learning in networks. Consequently, as a result of our attempts to shore up the mathematical foundations of our work, we developed an adaptive unit that performed much better in networks than any of the others we had tried.

During the period covered in this report, a number of other research groups became interested in the problem of learning nonlinear associative mappings by layered networks, or more generally, the problem of learning by "hidden units." In addition to our own method using A_{R-P} units, two new methods were developed: the Boltzmann learning procedure of Ackley, Hinton, and Sejnowski [1] and the error back-propagation method of Rumelhart, Hinton, and Williams [44]. These methods attracted much public attention, especially the backpropagation method which Sejnowski and Rosenberg [45] used in a system called NETtalk that learns how to convert text to speech. Unlike Boltzmann learning, which applies to symmetrically connected networks, the error back-propagation method applies to networks without cycles (acyclic networks). Consequently, error back-propagation is more directly comparable to the A_{R-P} method than is Boltzmann learning.

We invested much effort in performing simulations to compare various methods for learning in layered networks, including the error back-propagation method, and the results are reported here. In the comparisons, we included methods that represent several different approaches including the most brute-force search method possible. We chose a learning task that was hard enough to make the brute-force search inefficient but not so hard that enormous amounts of CPU time were required. On this task, the 6 input multiplexer task (see Section 4), the error back-propagation method proved to be the fastest with a modified A_{R-P} method coming second and the unmodified A_{R-P} method third. We did not systematically apply these methods to a series of increasingly difficult learning tasks in order to assess how they "scale" to larger problems. Our experience and theoretical understanding suggest, however, that the ordering of performance observed on the multiplexer task would be preserved on more difficult tasks. The comparative simulations do establish that

both the error back-propagation and the A_{R-P} methods are very much better than a variety of more conventional search methods.

Although we have not extended the A_{R-P} convergence theorem, which applies to a single adaptive unit, to a network of A_{R-P} units, much theoretical insight into the behavior of A_{R-P} networks has been provided by a result proved by R. Williams (one of the developers of the error back-propagation method). Williams [61,62] has shown that under certain restrictions on the A_{R-P} rule, the *expected* change of any weight within an arbitrary acyclic network of A_{R-P} units is proportional to the gradient of the probability of reward for the entire network with respect to that weight. This result means that A_{R-P} networks do something similar to what error back-propagation networks do, but they use estimates of the gradient which can be determined without the need for explicit back-propagation.

Because a gradient is estimated by A_{R-P} networks, the following modified training procedure is suggested. Instead of updating weights after a single presentation of an input pattern and the generation of a single activity pattern, one can hold the input pattern constant for several time steps and accumulate a gradient estimate during the generation of several activity patterns. Updating the weights on the basis of this improved gradient estimate should improve learning rate. We report the results of simulations designed to test this hypothesis in Section 4.

Also reported here are results obtained from applications of layered-network methods to two different tasks requiring the learning of problem-solving strategies. The first task is the pole-balancing task that we have used in the past to demonstrate reinforcement learning under conditions of delayed reinforcement [13]. The second task is to learn how to solve the Tower of Hanoi puzzle using a method that is essentially the same as the method used in learning to balance the pole. Our earlier work with the pole-balancing problem assumed the existence of a representation for the system's state consisting of a large number of non-overlapping "boxes" produced by a pre-existing decoder. Given this representation, the task became one of filling in look-up tables. This simplified representation allowed us to separate representation issues from the issues of temporal credit-assignment. In the studies reported here, the pre-existing decoder is replaced by a layered adaptive network. This network

receives as input a vector of four real numbers giving the state of the cart/pole system. The network has to learn how to represent the state so that the system as a whole can successfully avoid failure. The layered network provides a kind of adaptive decoder. In order to accomplish this, the adaptive critic element and the associative search element of previous studies were combined with the error back-propagation method for learning in layered networks. The resulting system was able to learn appropriate mappings for the control actions and the internal evaluation, and it was demonstrated that the multilayer system dramatically outperformed a single layer system.

Much the same approach was taken with the Tower of Hanoi puzzle. The state of the puzzle was represented as a binary vector that acted as input to two layered networks, one of which was responsible for forming an informative evaluation function, and the other of which was responsible for forming the correct mapping from puzzle states to actions (moving the disks). This system consistently learned to solve the puzzle using the minimum number of moves. This example allowed us to discuss the relationship between our strategy learning methods and an adaptive production system that has been applied to this puzzle [31].

In the concluding section of this report, I place our results in perspective by discussing their relationship to more conventional engineering methods. I also discuss directions in which I think it will be profitable to continue the development of these methods.

SECTION 2

THE ASSOCIATIVE REWARD-PENALTY UNIT

We developed a learning rule that we call the *associative reward-penalty*, or A_{R-P} , rule [7,6,9,8]. This rule, which can be implemented by a neuron-like adaptive unit that we call an A_{R-P} unit, is a refinement of similar learning rules that we had studied earlier. We devised it by combining aspects of algorithms for stochastic learning automata with aspects of algorithms for pattern classification or system identification. As a result of this hybrid nature, this method differs in critical ways from the methods, such as the perceptron and Widrow/Hoff LMS methods, that have become widely used in connectionist systems (for details, see Ref. [6]). I first give an informal description of the A_{R-P} learning rule, after which I specify it more formally and define the task it was devised to solve.

The A_{R-P} rule is an embellishment of Thorndike's [52] "Law of Effect":

Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond. (p. 244)

Although a literal interpretation of this "law" has numerous difficulties with respect to animal learning data, it remains a principle whose *basic* features have considerable,

but not uncontested, validity [33]. The A_{R-P} rule implements the basic idea of the Law of Effect, but it was necessary to add a number of refinements in order to make it work correctly.

Each situation referred to in the Law of Effect corresponds to an input vector, or key, that is received as input by an A_{R-P} unit. From this input vector, the unit determines an "activation level" which is the weighted sum of the components of the input vector, where the weights make up the unit's current weight vector. The unit then determines its action by comparing its activation level with a randomly varying threshold, "firing" (action = 1) when the activation exceeds the current threshold value, and otherwise not firing (action = 0). The noise in the threshold is such that when the activation is zero, the two actions are equiprobable; when it is positive, firing is the more likely action; and when it is negative, not firing is the more likely action. The activation level therefore determines the strength of the bond between the situation and the actions. As the weights change so as to increase the magnitude of the activation for specific input vectors, the bond between those vectors and the various actions increases—positive activation producing a bond between the input vector and firing; negative activation producing a bond between the vector and not firing.

The A_{R-P} learning rule causes the weight vector to change in such a way that if an action emitted in the presence of situation x yields an evaluation of "reward," the unit is more likely to produce the same action when x , or situations similar to x , occur in the future; in the case of penalty, weights change in such a way that the unit is more likely to produce the *other* action, when x , or situations similar to x , occur in the future. In order for this process to converge correctly to the actions that correspond to the highest probability of reward, it is necessary to change the weights asymmetrically in the cases of reward and penalty. Changes in the case of penalty must be much smaller than the corresponding changes would be in the case of reward. In the following sections, more technical descriptions of these ideas are presented.

The Associative Reinforcement Learning Task

The A_{R-P} learning rule is designed to solve what we call *associative reinforcement learning* tasks. In these tasks the learning system and its environment interact in a closed loop. At each discrete time step, or trial, t , the environment provides the learning system with a pattern vector, $x[t]$, selected from a finite set of vectors $X = \{x^{(1)}, \dots, x^{(m)}\}$, $x^{(i)} \in \mathbb{R}^n$; the learning system emits an action, $y[t]$, chosen from the finite set $Y = \{y_1, \dots, y_k\}$; the environment receives $y[t]$ as input and sends to the learning system a reward/penalty signal $r[t] \in \{\text{reward}, \text{penalty}\}$ that evaluates the action $y[t]$. The environment determines the evaluation according to a map $d : X \times Y \rightarrow [0, 1]$, where $d(x, y) = \text{Pr}\{r[t] = \text{reward} \mid x[t] = x, y[t] = y\}$. Ideally, one wants the learning system eventually to respond to each input vector $x \in X$ with action y_x with probability 1, where y_x is such that $d(x, y_x) = \max_{y \in Y} \{d(x, y)\}$.

As pointed out in Ref. [7], in the case of a single, nonzero input vector, this task reduces to the task usually studied by learning automaton theorists (which, according to the terminology used here, is a *nonassociative* reinforcement learning task); see Section 2 and Ref. [36]. On the other hand, in the case of two actions ($|Y| = 2$) the task reduces to a conventional formulation of supervised learning pattern classification (see [10]) if for each $x \in X$, $d(x, y_1) + d(x, y_2) = 1$. This restriction (assuming it is known to hold) implies that feedback received from performing one action provides information about the other action. This makes the task much easier and allows conventional supervised learning pattern-classification algorithms (slightly modified) to succeed (see Ref. [7] for details).

The A_{R-P} Learning Rule

The A_{R-P} rule's action selection method is parameterized at step t by a weight vector $w[t] \in \mathbb{R}^n$:

$$y[t] = \begin{cases} 1, & \text{if } w[t]^T x[t] + \eta[t] > 0; \\ 0, & \text{otherwise;} \end{cases} \quad (2.1)$$

where $w[t]^T x[t]$ is the inner product of $w[t]$ and $x[t]$, and the $\eta[t]$ are independent identically distributed random variables, each having distribution function Ψ .

According to Equation 2.1, the action probabilities at step t are conditional on the input vector in a manner determined by the parameter vector $w[t]$. In particular

$$p^{0x}[t] = Pr\{y[t] = 0 | x[t] = x\} = Pr\{w[t]^T x + \eta[t] \leq 0\} = \Psi(-w[t]^T x), \quad (2.2)$$

and

$$p^{1x}[t] = Pr\{y[t] = 1 | x[t] = x\} = 1 - p^{0x}[t]. \quad (2.3)$$

If, for example, each random variable $\eta[t]$ has zero mean, then when $w[t]^T x = 0$, the probability that each action is emitted given input vector x is .5; when $w[t]^T x$ is positive, action $y[t] = 1$ is the more likely action; and when $w[t]^T x$ is negative, action $y[t] = 0$ is the more likely.¹ As $|w[t]^T x|$ increases for all $x \in X$, the mapping Equation 2.1 approaches a deterministic linear discriminate function.

The parameter vector is updated according to the following equation:

$$w[t+1] - w[t] = \begin{cases} \rho[t](y[t] - p^{1x}[t])x[t], & \text{if } r[t] = \text{reward;} \\ \lambda \rho[t](1 - y[t] - p^{1x}[t])x[t], & \text{if } r[t] = \text{penalty;} \end{cases} \quad (2.4)$$

where $0 \leq \lambda \leq 1$ and $\rho[t] > 0$.

In the case of reward, according to Equation 2.4, w changes so that the probability of the action chosen, conditional on the current input vector, moves toward 1 (if $y[t] = 1$ then w changes so that p^{1x} approaches 1; if $y[t] = 0$ then p^{1x} decreases toward 0, which means that the probability of producing action 0 increases). In the case of penalty, on the other hand, w changes so that the probability of the action *not* chosen, conditional on the current input vector, moves toward 1. Note that the parameter λ in Equation 2.4 determines the degree of asymmetry in the magnitude of the weight change for these two cases.

It is shown in [7] that the A_{R-P} rule reduces under various restrictions to more conventional learning methods. It reduces to the two-action (nonassociative) linear

¹This version of the A_{R-P} rule differs from that given in Refs. [7,8,6] in that the actions are 0 and 1 instead of -1 and 1. The weight-update rule given below is altered so that the two versions are exactly equivalent. The 0/1 form allows the notation to be a bit simpler.

reward- ϵ -penalty ($L_{R-\epsilon P}$) learning automaton rule [36] when each $\eta[t]$ in Equation 2.1 is uniform in the interval $[-1, 1]$, the input pattern is constant and nonzero over time steps ($x[t] \equiv \hat{x} \neq 0$), and the initial parameter vector $w[1]$ is such that $w[1]^T \hat{x} \in [-1, 1]$. If additionally $\lambda = 0$, then the A_{R-P} rule reduces to the linear reward-inaction (L_{R-I}) rule [36]. On the other hand, when the A_{R-P} rule is made deterministic by letting $\eta[t] = 0$ for all t (i.e., the distribution function Ψ is the step function), then the A_{R-P} rule becomes the perceptron learning rule [42]. With a slight modification, the A_{R-P} rule can be reduced to the pattern-classification method introduced by Widrow and Hoff [59] (the adaline, or LMS, algorithm). Consequently, the A_{R-P} rule not only extends learning automata capabilities but also occupies the intersection of important classes of learning algorithms. Section 2 provides some background on learning automaton methods. The A_{R-P} rule is most closely related to the "selective bootstrap adaptation" method of Widrow, Gupta, and Maitra [58], to which it is compared in [7].

A convergence theorem is proven by Barto and Anandan [7] by extending to the associative case results proven by Lakshmivarahan [28,27]. It holds under the following conditions: (C1) the set of input vectors $X = \{x^{(1)}, \dots, x^{(m)}\}$, $x^{(i)} \in \mathbb{R}^n$, (C2) for each $x \in X$ and $t \geq 1$, $Pr\{x[t] = x\} > 0$; (C3) the independent, identically distributed random variables $\eta[t]$ in Equation 2.1 have a continuous and strictly monotonic distribution function Ψ ; and (C4) the sequence $\rho[t]$ in Equation 2.4 is such that $\rho[t] \geq 0$, $\sum_t \rho[t] = \infty$, $\sum_t \rho[t]^2 < \infty$. We can prove the following theorem:

Theorem. *Under conditions (C1)-(C4), for each $\lambda \in (0, 1]$, there exists a $w_\lambda^\circ \in \mathbb{R}^n$ such that the random process $\{w[t]\}_{t \geq 1}$ generated by the A_{R-P} rule in an associative reinforcement learning task converges to w_λ° with probability 1 (that is, $Pr\{\lim_{t \rightarrow \infty} w[t] = w_\lambda^\circ\} = 1$), where for all $x \in X$,*

$$\begin{aligned} Pr\{y = 1 | w_\lambda^\circ, x\} &> 1/2, \quad \text{if } d(x, 1) > d(x, 0); \\ &< 1/2, \quad \text{if } d(x, 1) < d(x, 0). \end{aligned}$$

In addition, for all $x \in X$,

$$\lim_{\lambda \rightarrow 0} Pr\{y = 1 | w_\lambda^\circ, x\} = \begin{cases} 1, & \text{if } d(x, 1) > d(x, 0); \\ 0, & \text{if } d(x, 1) < d(x, 0). \end{cases}$$

According to the usual performance criteria for learning automata [36], this result implies that for each $x \in X$, the A_{R-P} rule is ϵ -optimal. In fact, it implies a strong form of ϵ -optimality for each $x \in X$. It is highly unlikely that this result is the most general that can be proved about this class of learning rules (see [7]).

As is often done when using similar pattern-classification methods, in most of our simulations we hold $\rho[t]$ constant in order to increase learning speed even though a weaker form of convergence in this case has not yet been proven. We have not yet investigated elaborations of the A_{R-P} rule that reduce to recursive least squares methods based on the Newton's algorithm, but these have the possibility for showing improved convergence rates. We view condition (C1) that the set of input vectors is linearly independent as the most serious restriction required for the present theorem. It is likely that this restriction can be removed and a result proved that involves some form of operator pseudoinverse.

Simulation of a Single A_{R-P} Unit

In order to illustrate the performance of the A_{R-P} learning rule, we describe the results of simulating a single A_{R-P} unit in a simple associative reinforcement learning task that requires discrimination between two linearly independent, but non-orthogonal, input vectors. We use as a measure of performance the probability that the unit will receive reward on the average time step given its current parameter vector. We denote this $M[t]$ when computed based on the parameter vector $w[t]$:

$$\begin{aligned} M[t] &= \sum_{x \in X} \xi_x |Pr\{r[t] = 1 | x[t] = x\}| \\ &= \sum_{x \in X} \xi_x [d(x, 1)p^{1x}[t] + d(x, 0)p^{0x}[t]], \end{aligned}$$

where ξ_x is the probability that input pattern x occurs on any trial. This measure is maximized when the optimal action for each input pattern occurs with probability 1, in which case it is

$$M_{\max} = \sum_{x \in X} \xi_x \max\{d(x, 1), d(x, 0)\}.$$

The distribution function of the random variables used in all the simulations described here is the logistic distribution given by $\Psi(s) = 1/(1 + e^{-s/T})$, where T is a parameter. This is a sigmoidal function that is similar to a normal distribution function but is easier to evaluate. It is also used in the studies of statistical cooperativity (e.g., Ref. [24,1]), where T is the "computational temperature" of the system. As T approaches zero, the distribution function approaches a step function, which means that the A_{R-P} unit more closely approximates a deterministic system. Given this distribution function, the probability $p^{1x}[t]$ in Equation 2.4 is as follows (from Equations 2.2 and 2.3):

$$\begin{aligned} p^{1x}[t] &= 1 - p^{0x}[t] \\ &= 1 - \Psi(-w[t]^T x) \\ &= 1 - [1/1 + e^{w[t]^T x/T}] \\ &= \Psi(w[t]^T x). \end{aligned}$$

In all simulations presented here, we set $T = .5$.

In the first simulation the input vectors are: $x^{(1)} = (1,0)^T$ and $x^{(2)} = (1,1)^T$, which are linearly independent but not orthogonal. These vectors are equally likely to occur on each trial ($\xi_{x^{(1)}} = \xi_{x^{(2)}} = .5$). The weight vector, w , is zero at the start of each sequence of trials, which makes the actions initially equiprobable for both input vectors. The reward probabilities implemented by the unit's environment are given by the following table:

x	$d(x, 0)$	$d(x, 1)$
$x^{(1)}$.6	.9
$x^{(2)}$.4	.2

Thus it is optimal for the learning system to respond to $(0,1)^T$ with action 1 to obtain reward with probability .9, and to respond to $(1,1)^T$ with action 0 to obtain reward with probability .4. Therefore, in this task $M_{\max} = (.9 + .4)/2 = .65$, and the initial overall reward probability is $(.6 + .9 + .4 + .2)/4 = .525$. Note that any nonassociative learning automaton algorithm will be able to achieve a reward probability of at most $(.9 + .2)/2 = .55$ by learning to perform the action 1 at all times. Also note that

for each input x , the reward probabilities are either both greater than .5 or both less than .5, making this task considerably more difficult than one with the reward probabilities placed above and below .5 for each x .

Figure 1a shows results of simulating an A_{R-P} unit in this task with three different values of λ : .01, .05, and .25. We held the parameter $\rho[t]$ at the value .5 for all t . Plotted for each trial t is the average of $M[t]$ over 100 runs, where a run is a sequence of 5000 trials. The dashed lines show theoretical asymptotic performance levels for the three values of λ (if $\rho[t]$ were decreasing according to (C4)). Note that this asymptote approaches the optimal performance level .65 as λ decreases and that the learning rate decreases as λ decreases. The average final parameter vectors for $\lambda = .01, .05$, and .25 are respectively $(2.99, -4.04)^T$, $(2.73, -3.08)^T$, and $(1.91, -1.71)^T$. Figure 1b shows a plot of $M[t]$ for one of the runs contributing to the average shown in Fig. 1a for $\lambda = .05$. Although this task involves only two-dimensional pattern vectors, it illustrates the essential difficulties of learning to discriminate between patterns that are similar by virtue of sharing a subset of feature values.

A_{R-P} Units and Stochastic Learning Automata

The theory of learning automata originated with the independent work of the Soviet cybernetician Tsetlin [55], mathematical psychologists studying learning [16,19], and statisticians studying sequential decision problems (e.g., the "n-armed bandit problem" [41]). Although this theory has an extensive modern literature in engineering (reviewed in [36]), there has been very little cross-fertilization between this theory and neural-network research. In this subsection I briefly describe this theory, contrast it with the theory of supervised pattern classification, and describe how learning rules like the A_{R-P} rule can be seen as a synthesis of aspects of these theories.

Figure 2 shows a learning automaton interacting with an environment. At each step in the processing cycle, the automaton randomly picks an action from a set of possible actions, $Y = \{y_1, \dots, y_k\}$, according to a vector of action probabilities,

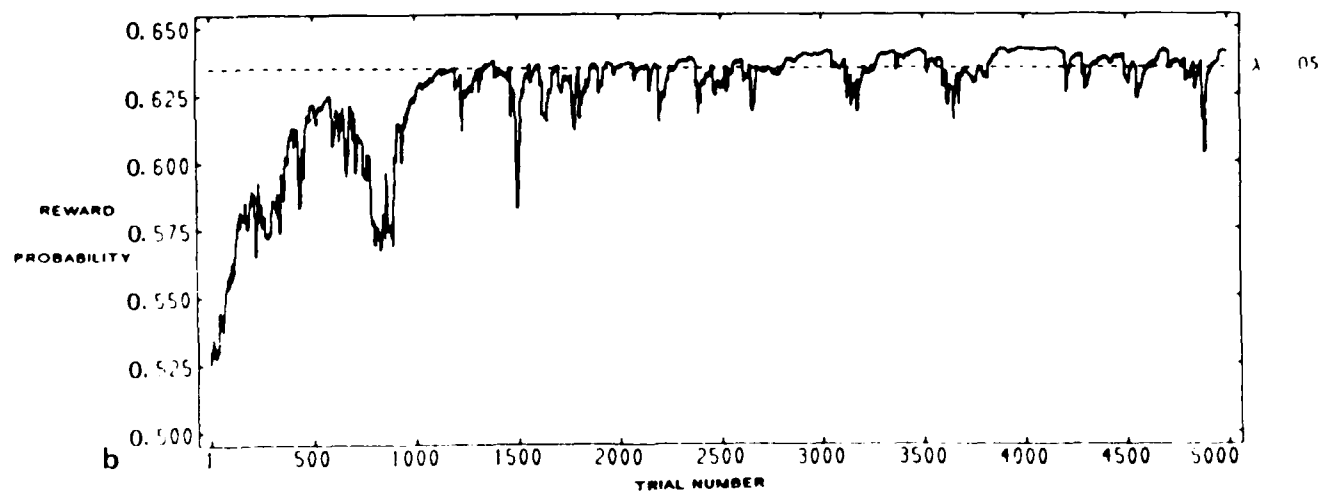
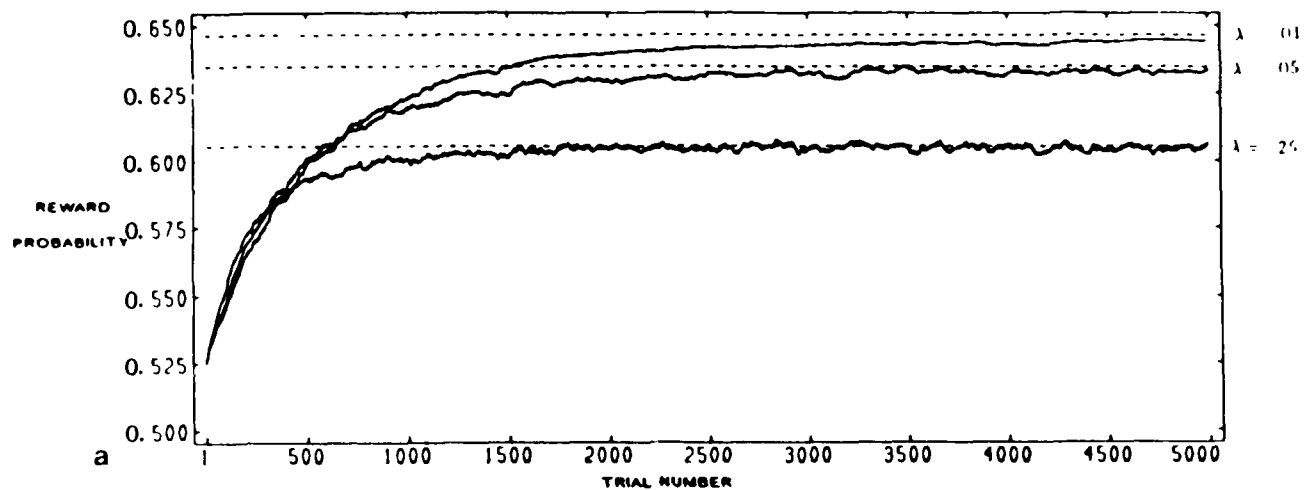


Figure 1: Simulation Results for a Single A_{R-P} Unit

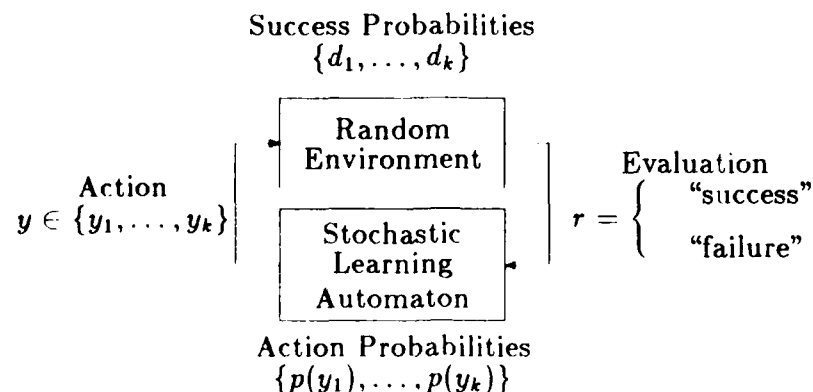


Figure 2: Stochastic Learning Automaton Interacting with a Random Environment.

$P = \{p(y_1), \dots, p(y_k)\}$. The environment then evaluates that action by selecting an evaluation signal that it transmits back to the automaton. Figure 2 shows the case in which the evaluation, r , is either “success” or “failure” and is selected according to probabilities $\{d_1, \dots, d_k\}$, where $d_i = \text{prob}\{\text{success}|y_i\}$ (other formulations allow a countable number or a bounded continuum of evaluations). Upon receiving the evaluation, the automaton updates its action probabilities as a function of its current action probabilities, the action chosen, and the environment’s evaluation of that action. Beginning with no knowledge of the environmental success probabilities, the objective of the automaton is to improve its expectation of success over time. Ideally, it should eventually choose action y_i with probability 1, where $d_i = \max\{d_1, \dots, d_k\}$. Many different algorithms have been studied under a number of different performance measures, and many convergence results have been proven [36].

Theorists have become increasingly interested in the collective behavior of learning automata. Figure 3 shows collections of N learning automata interacting with an environment. In Fig. 3a, each automaton receives a different evaluation signal that depends, in general, on the actions of all N automata. This models the situation in which the automata have differing, and possibly conflicting, interests. This is a game decision problem. In contrast to the problems studied in classical game theory,

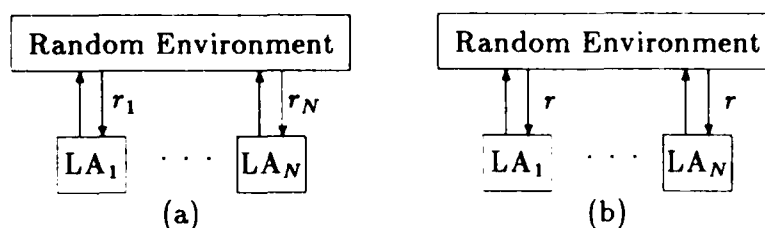


Figure 3: (a) The game problem. (b) The team problem.

the automata operate in total ignorance of the payoff structure of the game and the presence of the other automata. In the case of zero-sum games (games of pure conflict), theoretical results show that when employing certain algorithms, the learning automata converge to the game's solution (if that solution involves pure strategies; see Refs. [29,30,56]).

Figure 3b shows a collection of learning automata in the team situation, which is the special case of the game situation in which the automata receive the same evaluation signal. In this case, the automata have a common goal but each automaton only has partial control over the evaluation. As in the case of games, the learning process in this case is incompletely understood, but a number of mathematical results have been proven, the strongest of which shows that certain stochastic learning automaton algorithms lead to monotonic increases in performance [37].

Comparing stochastic learning automata and the typical adaptive units used in theoretical neural-network research reveals several important differences. First, a typical neuron-like adaptive unit has multiple input pathways that carry patterned stimulus information. Such a unit might also have a pathway specialized for training, such as the pathway for the desired response of a Widrow/Hoff Adaline or a Perceptron unit. The learning process causes the unit to implement or approximate a desired mapping from stimulus patterns to responses. A learning automaton, on the other hand, only has a single input pathway for the evaluation signal. Learning either results in the selection of a single optimal action or a suitable action probability vector — no (nontrivial) mapping is produced. On this dimension of comparison, then, the usual adaptive units are doing something more sophisticated than are learning automata.

However, the usual adaptive unit requires an environment that directly provides either a desired response or a signed error that directly tells the unit what response it should have produced. In contrast, a learning automaton has to discover, in a stochastic environment, which action is best by sequentially producing actions and observing the results. Since there are no constraints on the success probabilities, information gained from performing one action provides no information about the consequences of the other actions. This can be a non-trivial problem even in the case of two possible actions and is fundamentally different from the supervised learning problem [18]. Therefore, in terms of the amount of information required for successful learning, a stochastic learning automaton implements a form of learning that is more powerful than the supervised learning performed by most neuron-like adaptive units.

Because typical network adaptive units and learning automata excel on different dimensions, it has been fruitful to study learning units that combine the capabilities of these two types of systems. The resulting units, such as A_{R-P} units, are able to learn mappings in the absence of explicit instructional information. This ability has implications for applications to learning control problems as discussed in Section 5. Units such as A_{R-P} units can also participate in team or game decision problems similar to those in which learning automata have been studied. Unlike nonassociative learning automata, however, these units can learn to act conditionally on information from a variety of sources, including other units in a collection. Consequently, collective behavior more complex than that produced by nonassociative learning automata can be produced by networks of units combining associative learning with reinforcement learning. The following quotation illustrates that Tsetlin [55] was similarly interested in more elaborate forms of collective behavior:

We have discussed very simple forms of behavior, and for this reason we limited ourselves to the simplest types of automata. The exchange of information among these automata takes place in the language of penalties and rewards. Although this language seems universal enough, it would, however, be interesting to also look at more complicated automata that possess some specialized language to communicate to other automata. Such automata are needed to describe more complex forms

of behavior. These more complex behavioral forms necessitate the use of much more diverse information. (p. 125)

To the best of my knowledge, there has been no systematic attempt to study the collective behavior of learning automata that communicate in this way.² Some of our research represents the beginning of this type of study as described in the next section.

²Recent work by Thathachar and Sastry [51] uses stochastic learning automata in an algorithm for supervised pattern classification. Although this algorithm combines pattern classification and learning automata in a very interesting manner, it does not involve mutually communicating learning automata.

SECTION 3

COOPERATIVE BEHAVIOR OF A_{R-P} UNITS

In this section, I present an overview of our studies of cooperating collections of A_{R-P} units. Since details of the simulations are provided elsewhere [6], I mainly discuss the significance of these results and their relationship to other lines of research: the collective behavior of stochastic learning automata, game and team decision theory, and other methods for learning in layered networks. In Section 7, I briefly discuss problems with scaling this approach to larger problems and suggest how they might be solved by means of modularity and local reinforcement.

Associative Search Networks and Team Decision Problems

Our early work with the networks we called *associative search networks*, or ASNs, stressed the ability of these networks to learn associative mappings in the absence of explicit instructional information [14,11]. Figure 4 shows an ASN. It differs from the usual single-layer associative memory networks discussed in the connectionist literature (e.g., Ref. [23]) because instead of having reference channels for specifying desired outputs of the units, it has a single channel for broadcasting a scalar evaluation signal to all of the network's units. We studied ASNs in associative reinforcement learning tasks [14,12,11]. The object of such a task is to construct the mapping that associates each key with the action (recollection) that yields the best possible evaluation from the environment. A basic assumption is that the network has no *a priori* knowledge about the environment's evaluation function. If a network can solve this task, then the associative mapping it constructs has exactly the same properties as the mappings learned by the usual associative memory networks. In this

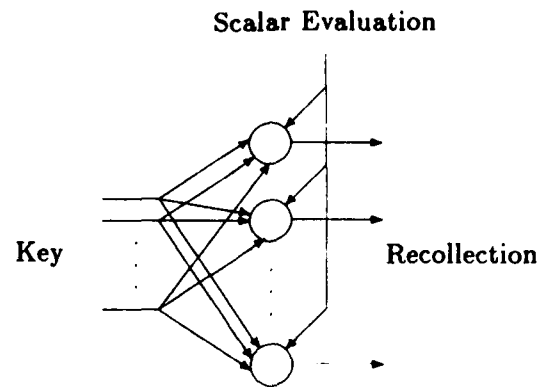


Figure 4: Associative search network.

case, however, the mapping would be formed in the absence of explicit instructional information. The scalar evaluation signal contains much less information than the reference vector required for storing information in the conventional case—it can be generated by an environment that can evaluate the behavior of the network, that is, the *collective* behavior of the network's units, but cannot specify the desired behavior of each individual component.

In addition to relating ASNs to associative memory networks, one can relate them to the teams of stochastic learning automata mentioned in Subsection 2 and shown in Fig. 3. An ASN in an associative reinforcement learning task is a generalization of a set of learning automata in a team decision problem. If one were to hold the input pattern to the ASN fixed for all time, the result would be the same as a team of nonassociative learning automata facing a team decision problem. Since all units receive the same reinforcement, they have no conflicts of interest. Consequently, the ability of an ASN to search for optimal patterns can be seen to arise from the cooperative activity of the adaptive units as each attempts to maximize its own performance. The ability of the adaptive units of an ASN to do this *conditionally* on information provided by the input patterns implies that the units cooperate to form associative mappings.

Layered Teams of A_{R-P} Units

A natural extension to the single-layer ASN is to add additional layers of A_{R-P} units. In these networks, units learn to act conditionally on information provided by other units in the network as well as information provided by the network's environment. As a result, layered networks can learn to implement nonlinear associative mappings. Suppose the network's environment presents stimulus patterns to the network by making the patterns' components available as input to some subset of the network's units. We call the units that receive this external stimulation the input units. The output signals of another subset of units are received by the environment, and patterns of these signals constitute the "overt" actions of the network. These are the output units, or to use the term of Hinton and Sejnowski [24], "visible units." The units that are not output units (including any input units that are not output units) we call the "hidden units" after Hinton and Sejnowski [24].¹ Suppose that the environment evaluates the activity of the visible units and broadcasts a reinforcement signal to all the units of the network.

How can a hidden unit improve its reward probability when its output cannot directly affect the environment? The only possibility is for it to assist visible units in increasing their reward probabilities; and this might be possible only by assisting intermediate units. For example, a hidden unit might adjust its weights in order to produce a signal A that another hidden unit combines with other information to produce a signal B, where signal B, in turn, allows a visible unit to make a required discrimination. The adaptive units must be able to discover how they can contribute to the common goal. We regard the linking up of units under these conditions to be a form of cooperation by which units coordinate their activities for mutual benefit.

¹Note that our use of these terms differs slightly from their usage by Hinton and Sejnowski and others. They replace each of our network input pathways with a specialized unit whose activation can be clamped to specific values by the network's environment, and they call these units visible units too. I have always preferred not to do this given my background in switching and automata theory.

A Minimal Layered Network of A_{R-P} Units

Figure 5 shows a network of two A_{R-P} units, u_1 and u_2 . Only u_1 receives stimulus patterns from the environment, and only the action of u_2 is available to the environment (u_1 is hidden; u_2 is visible). Suppose this network faces an associa-

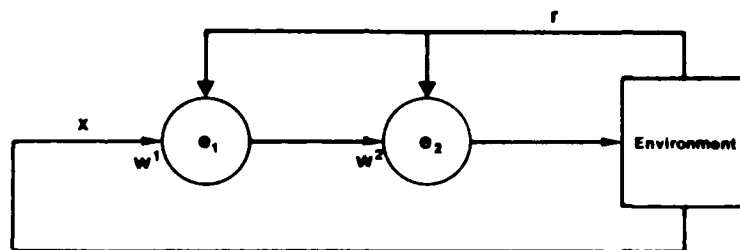


Figure 5: A Minimal Layered Network of A_{R-P} Units

tive reinforcement learning problem in which the network's output, the output of u_2 , affects the reward probability in a manner that depends on the stimulus pattern presented to u_1 . Both units receive the same reinforcement signal. If there were no means for u_1 to communicate with u_2 , the units would be capable of achieving only limited reward frequencies. The action of u_2 influences the reinforcement of both units, but in the absence of a communication link, u_2 remains blind to the discriminative stimulus and therefore cannot learn to respond selectively in a discrimination task. On the other hand, in the absence of a communication link, u_1 can sense the discriminative stimulus but cannot influence the reinforcement received. The complementary specialties of the two units have to be combined in order for each to attain optimal performance. In simulating this situation, we arranged for the action of u_1 to potentially influence u_2 by providing an interconnecting pathway with an initial weight of zero. If this weight can be adjusted properly, the network can respond correctly. However, the correct value of the interconnecting weight depends on how u_1 has learned to respond to its input. Conversely, the correct behavior of u_1 depends on the value of the interconnecting weight, that is, on how u_2 has learned to respond to its input signals. Thus the two units must adapt simultaneously in a

tightly-coupled cooperative fashion in order to maximize reward frequency.

To be more specific, we set up the simulation in the following way. Each unit is provided with a constant input (equal to 1) to allow its threshold to vary and one other input pathway. We regard only this second stimulus component as the stimulus pattern x , treating the constant input as part of a unit's internal mechanism. Each unit of the network in Fig. 5 can therefore receive the input "pattern" 0 or 1, where for u_1 it is generated by the network's environment, and for u_2 it is the output of u_1 . The reward probabilities implemented by the network's environment are given by the following table:

x	$d(x, 0)$	$d(x, 1)$
0	.9	.1
1	.1	.9

Table entry $d(x, y)$ is the network reward probability given that u_1 receives x as input and u_2 responds with y as output, that is, given that the network as a whole responds to x with y . Thus it is optimal for the network to respond to $x = 0$ with action 0 to obtain reward with probability .9, and to respond to $x = 1$ with action 1 to obtain reward with probability .9. In this task $M_{\max} = (.9 + .9)/2 = .9$, and the initial overall reward probability (with all weights zero) is $(.9 + .1 + .1 + .9)/4 = .5$. Note that if the network fails to discriminate by responding identically to all input patterns, the overall reward probability is $(.9 + .1)/2 = .5$.

There are two ways the network can solve this problem. Let us denote the weights associated with u_i 's (nonconstant) input pathway w^i , $i = 1, 2$. In the first solution, u_1 learns to fire only when stimulus $x = 1$ is present by setting its threshold high (i.e., setting its threshold weight negative) and setting w^1 positive. Unit u_2 does the same thing—sets its threshold high and w^2 positive—so that it fires only when stimulated by u_1 's firing. Consequently, the network as a whole fires only when $x = 1$. In the second solution, u_1 learns to fire at all times *except* when stimulus $x = 1$ is present, and u_2 learns to fire at all times *except* when u_1 fires. Then when u_1 is silent in response to $x = 1$, u_2 is disinhibited and so fires.

In simulating a trial with this network, and with all the networks to be described, the environment first presents a stimulus pattern to the network, and then proceed-

ing from the input side of the network, we sequentially compute the output of the successive units so that their actions are available as input to "downstream" units. This is possible because the networks described here do not have recurrent connections. When the network's overt action is generated, the environment produces the reinforcement signal, and all the units update their weights. We view the weight modifications as occurring simultaneously for all units, although this is actually done sequentially by the computer program.

Figure 6 shows the behavior of the network for a typical sequence of 500 trials with $\lambda = .04$ and $\rho = 1.5$. Figure 6a shows the evolution of the behavior of u_1 in terms of two graphs. The first shows the conditional probability that u_1 fires ($y_1 = 1$) given that its (nonconstant) input is 0, and the second shows the same thing for input 1. Both of these probabilities start at .5 since the weights are initially zero, and they change in approximately the same way for about the first 50 trials. This means that during these trials the unit is experimenting with firing and not firing in the presence of both input signals. At this point the two conditional probabilities show the beginning of differentiation between the two cases, which becomes unequivocal by about trial 80. From then on, with a few brief exceptions, u_1 has a high probability of firing in response to an input of 1 and a low probability of firing in response to an input of 0. Figure 6b shows the evolution of the mapping implemented by u_1 and u_2 acting together by showing the probability that u_2 fires ($y_2 = 1$) for the different values of the network input x (not for the values of u_2 's local input). Since the network learns to respond correctly, u_2 learns to remain silent unless excited by u_1 's activity; that is, the first solution is formed in which both w_1 and w_2 become positive and both units set high thresholds. Figure 6c shows the evolution of the overall performance measure M_t . Figure 6d is a histogram of the number of trials required to reach a criterion of 98% of M_{\max} for each of 100 sequences of trials. In all sequences the network reached this criterion before 1500 trials. In 45% of the sequences, the network produced the first solution; in the remainder it produced the second.

A series of two units in a discrimination task provides one of the simplest examples we could devise to demonstrate statistical cooperativity of self-interested units. It is

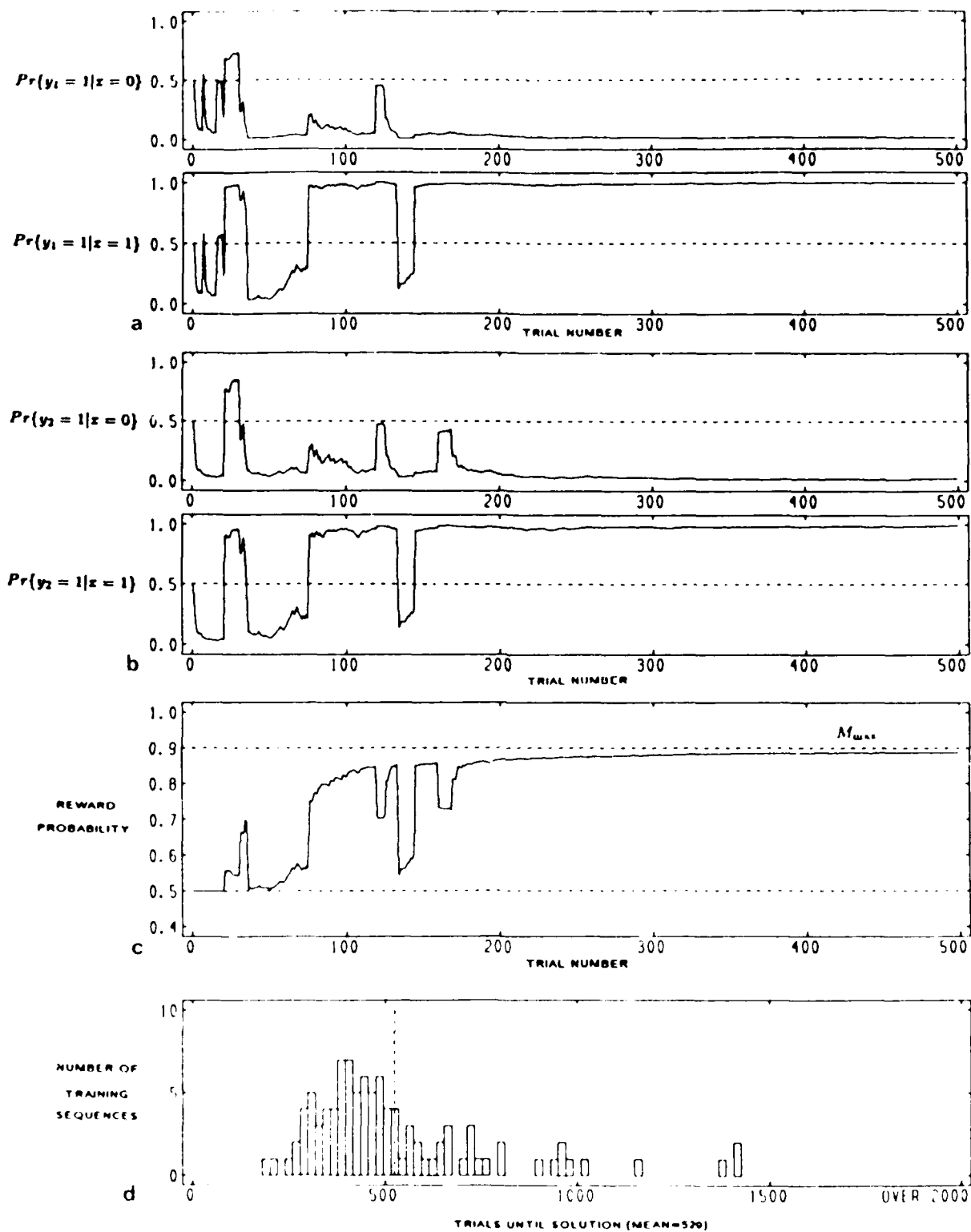


Figure 6: Simulation Results for the Two-unit Network.

clear that the A_{R-P} units effectively form a link that permits them to obtain higher reward rates than they could attain if they were to act independently. Moreover, a unit contributes to the formation of this link only because doing so furthers its interests. We interpret this as a form of cooperativity in the literal game-theoretic sense. One may regard the link as a "binding agreement" by which the units form a coalition for mutual benefit. We have simulated series of 3, 4, and 5 units with appropriate connections being made in all cases, although learning slows considerably as the depth of the network increases. Although the discrimination required in these tasks is not difficult, the necessity to construct a long chain of connections that faithfully transmits the discriminative stimulus is quite difficult. The correct behavior for any unit depends on the behavior implemented by all the other units so that the solution cannot be constructed from stable solutions to subtasks.

The XOR Task

In the task just described, cooperative learning is required only because the network lacks a direct pathway from input to output. The task itself is easily within the capabilities of a single unit. Here we illustrate the simplest example of a task that cannot be solved by a single linear threshold unit, or any single-layer network of them. In this problem the hidden unit is needed not just to transmit a discriminative stimulus to the visible unit; the hidden unit must learn to respond to particular configurations of its stimulus components in order to create a signal that the visible unit needs to behave properly. In our simulation, a network of two A_{R-P} units is placed in a task requiring it to form the two-component exclusive-or mapping. The network has a single hidden unit, u_1 , and a single visible unit, u_2 , which are connected as shown in Fig. 7. The stimulus patterns are all the two-component binary vectors: $x^{(0)} = (0, 0)$, $x^{(1)} = (0, 1)$, $x^{(2)} = (1, 0)$, $x^{(3)} = (1, 1)$. These patterns are equally likely to occur on any trial. Each unit also has a constant input and a threshold weight.

The reward probabilities are given by the following table:

x	$d(x, 0)$	$d(x, 1)$
(0, 0)	.9	.1
(0, 1)	.1	.9
(1, 0)	.1	.9
(1, 1)	.9	.1

Table entry $d(x, y)$ is the reward probability given that the network receives x as input and responds with action y . The optimal reward probability is $M_{\max} = .9$, which is obtained when the action of the visible unit is the exclusive-or of the pattern components, that is, when u_2 fires when one or the other, but not both, stimulus components are present. It must also not fire when both components are absent. A single A_{R-P} unit can be correct for at most three of the four cases, yielding a reward probability of .7, since weights do not exist that allow a single linear threshold unit to respond correctly to all four stimuli (see Duda and Hart, 1973, or Minsky and Papert, 1969). However, the performance of the network of Fig. 7 can approach M_{\max} if the hidden unit learns to respond only to the fourth case and the visible unit takes advantage of this signal to "debug" its responding. This can happen in several ways depending on whether the hidden unit learns to turn on or off for the fourth case.

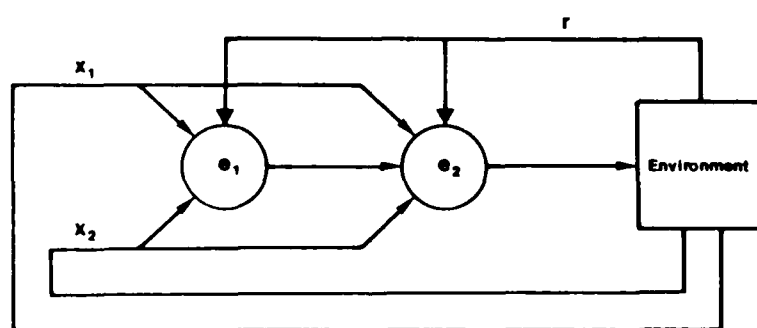


Figure 7: Network for the Exclusive-Or Task.

Figure 8 shows performance of the two-unit network for a typical sequence of 5000

trials with $\rho = 1.5$ and $\lambda = .08$. In Fig. 8a are graphs showing how the output probabilities of the visible unit develop for each input pattern; Fig. 8b shows the analogous information for the hidden unit; and Fig. 8c shows the overall performance of the network as a function of the trial number. The visible unit quickly learns to respond correctly to all patterns except $x^{(1)} = (0, 1)$ (Fig. 8a), causing the network performance to level off near .7 (Fig. 8c). Eventually ($t \approx 1400$) the hidden unit comes to respond reliably to $x^{(1)}$ and to reliably not respond to any other pattern (Fig. 8b). At the same time, the visible unit begins to be excited by the hidden unit's signal so that its output tends to be correct more frequently for all four patterns (Fig. 8a). Once this mutually beneficial relationship between u_1 and u_2 begins, it quickly develops until almost perfect performance is achieved (the theoretical asymptote is .892 for this value of λ). It is clear that this is a cooperative process.

Figure 9 shows a histogram of the number of trials until a criterion of 95% of M_{\max} is attained for each of 100 sequences of trials. The average number of trials until criterion is 3501, or about 875 trials for each stimulus pattern. In all of the sequences the network reached this criterion before 15,000 trials.

The Multiplexer Task

The network shown in Fig. 10 has six input pathways and a single principal output pathway (from unit 5). There are 39 weights to adjust: one associated with each of the pathway intersections and one threshold weight for each unit. The reward contingencies implemented by the network's environment force the network to learn to realize a multiplexer circuit in order to obtain optimal performance. A multiplexer is a device with k address input pathways and 2^k data input pathways (here $k = 2$), each of which is associated with a distinct k -bit address. Given a pattern over the address pathways, i.e., an address, a multiplexer's output is equal to whatever signal (0 or 1) appears on the data pathway associated with that address. It therefore routes signals from different input pathways to a single output pathway depending on the "context" provided by the pattern over the address pathways. If we call the address components a_1 and a_2 and the data components d_1, d_2, d_3 , and d_4 , a minimal

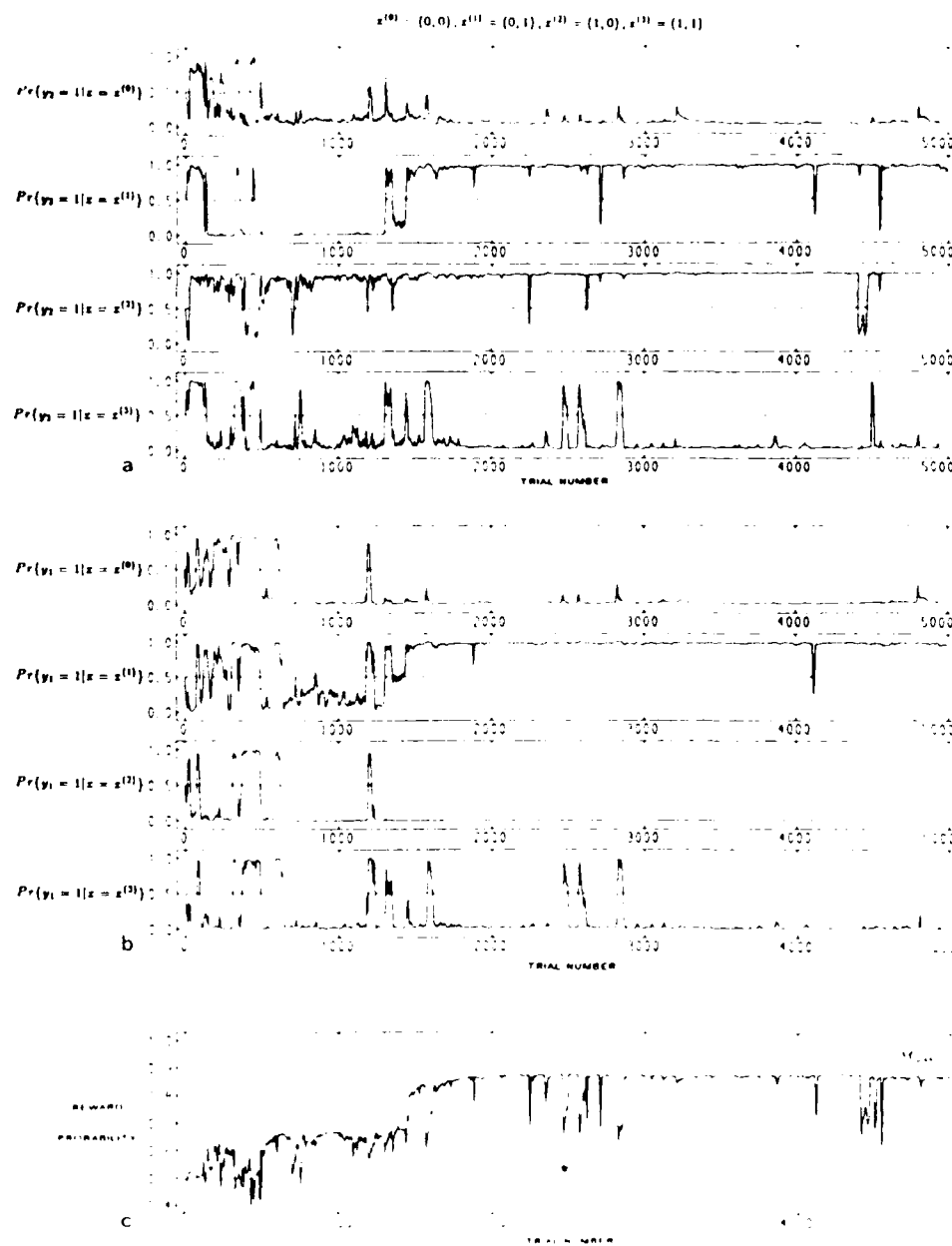


Figure 8: Behavior of Network Units in a Typical Sequence of 5000 Trials in the Exclusive-Or Task.

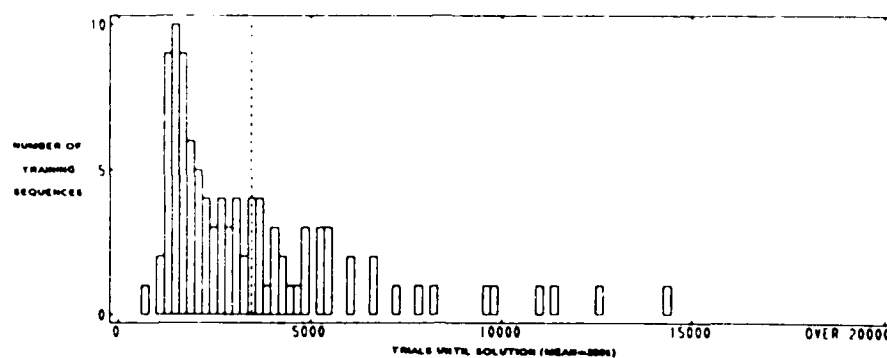


Figure 9: Histogram of Trials to Criterion for 100 Sequences of Trials in the Exclusive-Or Task.

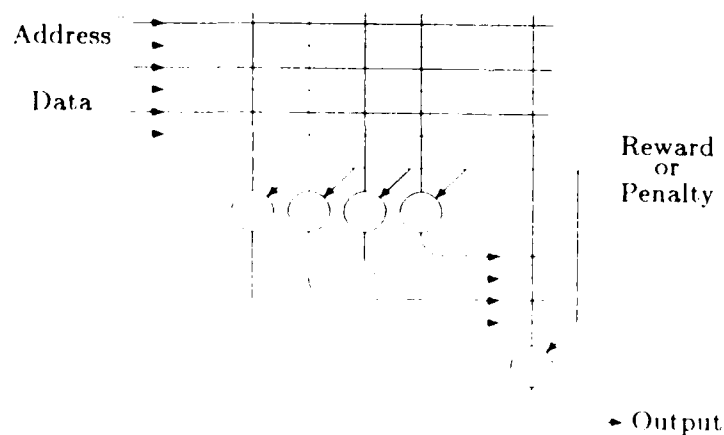


Figure 10: Layered Network for the Multiplexer Problem.

logical expression for the multiplexer function is

$$\bar{a}_1 a_2 d_1 \vee \bar{a}_1 a_2 d_2 \vee a_1 \bar{a}_2 d_3 \vee a_1 a_2 d_4.$$

There are a total of 2^6 , or 64, input patterns.

For each of the 64 possible input patterns, we rewarded each unit of the network with probability 1 if the visible unit (unit 5) produced the correct output, and we penalized each unit with probability 1 otherwise. The input patterns were chosen randomly for presentation to the net. All of the units implement the A_{R-P} algorithm with $T = .5$ except for the visible unit (unit 5) which uses $T = 0$ (and therefore essentially uses the perceptron algorithm; see Section 2). Fig. 11 is a histogram of the number of trials required for the network to respond 99% correctly for 1000 consecutive trials in each of 30 sequences of trials with $\rho = 1$ and $\lambda = .01$. The average number of trials required is 133,149, or about 2080 presentations of each stimulus pattern. In every sequence the network reached the criterion before 350,000 trials.

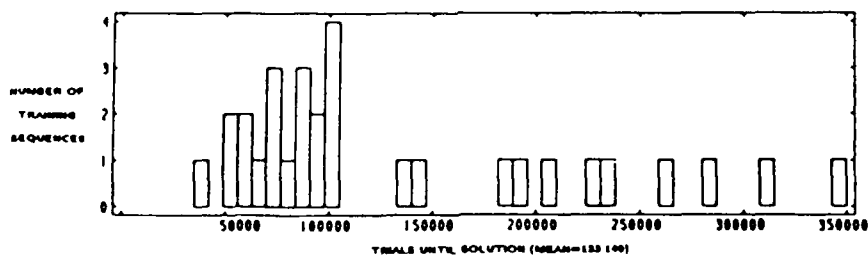


Figure 11: Histogram of Trials to Criterion for the Multiplexer Task.

This task illustrates some of the computational sophistication that can arise with the formation of nonlinear functions. Linear threshold functions can exhibit only a very restricted form of context sensitivity: contextual information can bias activation one way or the other, effectively raising or lowering a threshold. Nonlinear context sensitivity, on the other hand, can result in the complete alteration of behavior as a function of contextual information. The exclusive-or task described in Section 3

illustrates this in the simplest form, where one stimulus component can be regarded as switching the processing of the second stimulus component between the identity and inversion functions. The multiplexer illustrates a more extreme form by which the contextual information provided over the address pathways completely alters the set of signals to which the principal unit is sensitive.

Discussion— A_{R-P} Networks and Gradient Descent

Not long after we began experimenting with networks of A_{R-P} units, Rumelhart, Hinton, and Williams [44] presented an error back-propagation method for learning in layered networks that has since become well-known. This method is described in Section 4, where its performance is compared with that of several other methods including the A_{R-P} method. This error back-propagation method is now deservedly popular since it is simple to understand and outperforms other methods for learning in layered networks, including ours based on A_{R-P} units. What is most interesting here is that the error back-propagation method together with a theoretical result of Williams [61,62] sheds much light on the collective behavior of A_{R-P} units in layered networks. In fact, it is not too misleading to regard A_{R-P} networks as performing a kind of stochastic approximation to the back-propagation method (although this is not strictly true for several reasons to be discussed).

The error back-propagation method is a gradient descent procedure in weight space. The remarkable result is that information about how to step in weight space to minimize (or maximize) a global network performance criterion can be obtained locally in the networks. In the case of the back-propagation algorithm, this information—the partial derivative of the performance criterion with respect to each weight—is obtained through a complex process in which error signals are transformed and passed backward through the network. Another way for a unit to determine what steps to take in weight space is for it to determine the derivative of the performance criterion with respect to its activity by varying its output and observing how the global performance changes as a result. Given this estimate, the unit can then correctly determine how to change its weights.

More specifically, suppose the units are deterministic, and that a given hidden unit can vary its output around its current value while the outputs of all of the other units are frozen at their current values. By observing the consequences of this variation on the performance criterion, the unit can determine the gradient of the criterion with respect to its output at the current point in weight space. From this it can easily determine the criterion's derivative with respect to its weights, and so can alter them appropriately. Now each unit in turn can do this with the other units frozen. If a unit's new weights are not put into place until all the units have varied their outputs, the result will be a step in weight space according to the gradient of the criterion. This process, which is reminiscent of, but different from, the Boltzmann relaxation process, would work but has obvious shortcomings since some outside agency would have to orchestrate the process and it would be quite slow.

But can the units vary their outputs *simultaneously* and observe the consequences to achieve the same result? This could be made to work if the units independently influenced the criterion function, but it is difficult to see how it could be done if these influences are not independent, which is the only case of real interest. It turns out, however, that it is possible for interacting units to simultaneously vary their outputs to obtain an *estimate* of the appropriate gradient. This is essentially what happens in networks of A_{R-P} units. Williams has shown [61] for an arbitrary acyclic network of A_{R-P} units that if the parameter λ in Equation 2.4 is zero for each unit², then the expected direction of each weight change is proportional to the gradient of the global network reward probability. Consequently, each weight changes according to an unbiased estimate of the partial derivative of the global criterion function with respect to that weight. On any particular trial, the step in weight space actually taken may or may not amount to an improvement, but the trend will always be in the correct direction.

Thus, A_{R-P} networks (with $\lambda = 0$) provide a way of locally computing gradient information without the need for a complex back-propagation process. We have found that in practice such networks actually require λ to be nonzero (in fact, a

²We call units with $\lambda = 0$, A_{R-I} units, for *Associative Reward-Inaction* units: upon penalty, no weight changes occur.

positive value much smaller than ρ) in order to converge properly. Although the average direction of weight change is correct when $\lambda = 0$, the process can get stuck at suboptimal points in weight space because the units become deterministic too soon. Setting λ nonzero seems to prevent this from happening by eliminating all absorbing states from the stochastic process. Consequently, even after learning is complete, all the units retain a small amount (depending on the size of λ) of random variability in their behavior.

This view of A_{R-P} networks provides a link, albeit an approximate one, to the gradient descent procedure implemented by the Rumelhart et al. back-propagation method. The link is not exact for two reasons: 1) since A_{R-P} units are binary whereas back-propagation units have continuous outputs, the activity spaces in the two cases are different, and 2) the criterion functions in the two cases are different—in the A_{R-P} case it is the network reward probability whereas in the back-propagation case it is the total mean-square-error of the visible unit's activity. Nevertheless, the relationship between these two methods is useful in understanding the cooperative interaction that occurs in A_{R-P} networks. As one would expect from this relationship, the A_{R-P} method is slower than is the back-propagation method in terms of the number of stimulus pattern presentations. This is borne out in the comparative studies described in the next section. However, the A_{R-P} method does not require a back-propagation process to assign credit to the units. This could have advantages in terms of hardware implementation and in terms of biological plausibility. The relationship of the A_{R-P} method to gradient descent also suggests a modification of the A_{R-P} learning scheme, which we call the *batched* A_{R-P} method, that is described in Section 4.

SECTION 4

COMPARATIVE STUDIES OF LAYERED NETWORK LEARNING METHODS

In assessing any new approach to an old problem, it is necessary to compare the new method with ones that have been tried before. We therefore conducted simulation studies designed to compare a number of methods that have been proposed for learning in layered networks. We compared eleven such methods by applying each to the same learning task. We chose the multiplexer task (see Section 3) because it is difficult enough to show the advantages of the more sophisticated methods, but it is simple enough that reasonable amounts of CPU time are required for statistically significant comparisons. In this section I review the results obtained. Complete details are available in Ref. [4] from which this section is abstracted.

In the experiments to be described, the hidden-unit learning rule is the primary variable. The learning rule for the output unit is the same for most experiments. The perceptron learning rule [42] is used for the output unit since it is well-known and is relatively insensitive to the learning rate parameter ρ (so ρ would not have to be varied to optimize performance).¹ The network structure is as in Fig. 10. A step in the simulation of this system consists of the following. An input vector is selected by choosing one randomly, without replacement, from the set of all input vectors. Upon receipt of an input vector, the outputs of the hidden units are calculated, followed by the calculation of the output of the output unit. The output of the output unit is subtracted from the desired output. This error controls the perceptron learning rule as it is applied to the weights of the output unit, after which the particular learning method being tested in the hidden units is applied to the hidden

¹The application of the error back-propagation method [44] to the hidden units requires the use of a differentiable output function in the output unit, so a semilinear output function and learning rule were used in the output unit for the experiments with the error back-propagation method.

units' weights (although some methods, such as the direct-search methods, do not change the weights of the hidden units on every step). This completes one step in the simulation. Every input vector is presented once during the first 64 steps, and once again for every subsequent set of 64 steps, where the order of presentation is determined randomly.

The direct-search methods are presented first. These methods require no knowledge about the network other than the number of hidden-unit weights and their ranges of values. Following the direct-search methods, several error back-propagation methods are presented that involve the propagation of the output unit's error to the hidden units. Several reinforcement-learning methods are then presented, including the A_{R-P} method. A modification of one reinforcement-learning rule is considered that generates localized reinforcements to the hidden units by propagating information from the output unit back to the hidden units. Finally, a mechanism is considered that treats hidden units that have not yet acquired a substantial influence on the output unit differently from those that have developed influence.

The behavior of each method depends on several parameters. A comparative study should guarantee that parameter values are used that are optimal for a given method to ensure the absence of bias in favor of one method over another. However, the time required to simulate the learning process in these experiments prohibited a thorough optimization of the parameter values. We were able to test an average of six different values over a broad range for each parameter, and when a method depends on more than one parameter, only one parameter was varied at a time. Note that this attempt to compare methods, where each is operating with optimal parameter values, does not address the important issue of the relative degree of robustness of the methods. Since the parameter values that are optimal depend on the learning task, it is possible that a learning method may excel at a particular task when using specific parameter values and yet perform badly on another task when using those same parameter values. On the other hand, a method that learns more slowly than other methods on a specific task may have a speed advantage over the other methods when applied, with the same parameter settings, to a *class* of tasks. The comparative studies reported here do not address this important issue.

Another important issue that is not addressed by these studies is the issue of scale-up. How do learning times grow as tasks get larger or more difficult? We did not apply the battery of learning methods to a series of increasingly difficult learning tasks.

Direct-Search Methods

Unguided Random Search

The simplest possible brute-force random search was included to provide some idea of how difficult the test learning task is. This method consists of randomly choosing new weight values for all of the hidden units in the network (using uniform probability density function); evaluating these weights by allowing the network to interact with its environment for a number of steps (denoted n); and remembering after each evaluation period the weight values receiving the best evaluation so far. Here we want to evaluate the current values of the weights by measuring how well the network can solve the task using the given weight values. The output unit continues to learn while the weights of the hidden units are held constant. The weights of the output unit are set to zero whenever new values for the hidden units' weights are generated.

The unguided random search was tested on the multiplexer task for several values of n . For each value of n , the results from 10 runs of 300,000 steps were collected. The final performance level of a run, ν , is the number of input vectors for which the network is incorrect when using the best set of weights found on that run (so $0 \leq \nu \leq 64$ and a purely random strategy of generating outputs would result in an average value of 32).

In addition to the performance level at the final step of each run, we determined the value of a measure of cumulative performance, μ , which for a single run is the sum of the number of errors made on every step. For a nonlearning, random strategy, errors would occur on an average of half of the steps, producing a value for μ of 150,000.

The results of the experiments are listed in Table 1, including the 99% confidence intervals of ν and μ . The unguided random search performed better than a nonlearning, random strategy for all values of n that were tried. The value of μ consistently declines as the parameter n increases. Recall that after every n steps, a new weight vector is generated that does not depend on previously-tested vectors, so there is no gradual improvement in performance as a run progresses. However, since the output unit is learning throughout each n step period, larger values of n result in better performance at the end of the n step period and better average performance over that period, which explains the inverse relationship of μ and n .

Table 1: Unguided Random Search on the Multiplexer Task

n	ν	μ
50	25.6 ± 2.78	$140,228 \pm 263$
100	22.7 ± 2.88	$134,397 \pm 128$
200	23.4 ± 3.80	$127,913 \pm 237$
400	18.0 ± 3.21	$122,209 \pm 176$
800	16.5 ± 2.66	$117,899 \pm 342$
1600	15.7 ± 3.22	$115,099 \pm 324$
3200	18.4 ± 5.23	$112,848 \pm 462$
6400	17.0 ± 4.29	$112,577 \pm 803$
12800	16.9 ± 3.96	$111,477 \pm 1,064$
25600	17.7 ± 3.71	$110,654 \pm 1,535$

The values of ν do not show that any one value of n is optimal. When n is 200 or less, significantly higher values of ν are obtained than when n is 400 or greater. In fact, for $n \leq 200$, performance is not significantly different from that of a single layer, for which $\nu \approx 24$.

A learning curve for the unguided random search on the multiplexer task was obtained by choosing the best value of n , which is 1600, and performing 30 runs of 300,000 steps each. This resulted in performance measures of $\nu = 17.0 \pm 2.93$ and $\mu = 115,062 \pm 229$ and the learning curve in Fig. 14 (the upper-most curve). On this and all subsequent graphs, an initial rapid drop appears from 0.5 errors per step to approximately 0.37 or 0.38. This is caused by the output unit learning as many correct responses as possible without using hidden units; a single unit given the input

vectors for the multiplexer task can learn the correct output for 40 of the 64 input vectors, resulting in an average of 0.375 errors per step.

Guided Random Search

There are obviously many ways of improving the unguided random search, all of which involve generating weight vectors that depend on the currently-best vector (or on a series of best vectors). We studied two methods: a guided random search and the polytope method described below. The guided random search differs from the unguided random search only in the manner of generating new weight vectors. Rather than being chosen according to a uniform probability density function, weight vectors are chosen from a unimodal probability density function (defined below) centered on the weight vector that is currently the best. This density function is symmetric about the currently-best vector, and the probability of selecting vectors decreases as the Euclidean distance from the currently-best vector increases. We used a density function based on the logistic distribution (see Ref. [4] for details). The method depends on two parameters: the number of steps between the generation of weight vectors, n , and the spread of the density function, τ .

As stated earlier, the amount of computer time required to perform these experiments prevented a systematic search for the optimal values of n and τ . However, we did perform two unidimensional searches by holding $\tau = 2$ while varying n , then varying τ while holding n at the value resulting in the best performance. For each parameter setting, results were averaged over 10 runs with each run lasting 300,000 steps. The results in Table 2 show that intermediate values of n are required to achieve good performance. However, unlike the results for the unguided random search, the cumulative performance measure, μ , also has a U-shape as n increases, providing evidence of a tradeoff between learning in the output unit (large n) and optimizing the weights of the hidden units by making more trials (small n).

Performance as a function of τ also has a U-shape — there appears to be an optimal value of τ in the range of 0.5 to 2 (as the value of τ increases, the probability density function approaches a uniform density function, and the behavior of guided random search approaches that of the unguided random search. The learning curve in Fig. 14

Table 2: Guided Random Search on the Multiplexer Task

n	ν	μ	τ	ν	μ
50	27.2 ± 3.76	$138,978 \pm 898$	0.1	18.9 ± 4.91	$106,894 \pm 6,204$
100	24.1 ± 3.61	$131,957 \pm 2,102$	0.2	17.1 ± 2.53	$109,454 \pm 4,897$
200	18.4 ± 3.94	$124,089 \pm 1,345$	0.5	14.9 ± 4.13	$105,343 \pm 6,124$
400	13.8 ± 3.76	$115,390 \pm 3,271$	1.0	11.4 ± 3.58	$102,583 \pm 6,128$
800	13.3 ± 4.73	$111,205 \pm 3,851$	2.0	12.5 ± 3.94	$106,818 \pm 2,524$
1600	13.1 ± 4.21	$106,544 \pm 3,314$	4.0	15.6 ± 2.83	$108,128 \pm 2,797$
3200	12.5 ± 3.94	$106,818 \pm 2,524$	8.0	15.0 ± 4.06	$108,498 \pm 3,066$
6400	16.5 ± 4.48	$108,225 \pm 2,413$			
12800	17.6 ± 5.29	$108,620 \pm 3,615$			

$\tau = 2$

$n = 3200$

was produced by averaging 30 runs of 300,000 steps each, using $n = 3200$ and $\tau = 1$. The resulting performance levels are $\nu = 13.1 \pm 2.36$ and $\mu = 103,866 \pm 3420$.

The Polytope Algorithm

Another method for directly searching the weight space is the Polytope Algorithm [21]. This method is often called the “simplex” method, not to be confused with the simplex method for linear programming. The polytope algorithm is a deterministic hillclimbing method that maintains a list of m weight vectors, ordered according to their evaluations. The m weight vectors are treated as vertices of a polytope in $m - 1$ -dimensional space, and new vectors are generated in a fashion designed to shift the polytope towards an optimum weight vector, taking large steps when progress is being made in improving the evaluation and taking smaller steps when it appears that the optimum has been approached. Since this is a deterministic hillclimbing method, it can get stuck at a local optimum, but it is good at following ravines. We included it in our study as an example of a reasonably sophisticated, deterministic, direct-search algorithm to complement the random methods presented above.

The polytope algorithm depends on the parameter m , the number of weight vectors maintained as vertices of the polytope, and the parameter n , the number of steps over which each weight vector is evaluated. Other parameters are ρ_r , ρ_c ,

and ρ_c , which determine the lengths of reflection, expansion, and contraction steps, respectively. Valid ranges for these parameters are $\rho_r > 0$, $\rho_e > 1$, and $0 < \rho_c \leq 1$. To reduce the number of experiments to a practical level, we did not attempt to find optimal values for ρ_r , ρ_e , and ρ_c , but set them to reasonable values. We did vary m and n , as shown in Table 3. The value of m was fixed at 20 while n varied, after which n was fixed at 1600, which gave the best value of ν , while m was varied. The results are again averages over 10 runs at 300,000 steps per run.

Table 3 suggests that the optimum value of n is between 400 and 3,200. The results are even less conclusive about the optimum value of m ; additional runs must be made to obtain performance averages with less variance. The values $n = 1600$ and $m = 10$ were used in 30 runs of 300,000 steps to obtain the learning curve in Fig. 14, resulting in $\nu = 14.2 \pm 2.09$ and $\mu = 94,977 \pm 3079$.

Table 3: Polytope Algorithm on the Multiplexer Task

n	ν	μ	m	ν	μ
200	20.8 ± 4.04	$118,780 \pm 6,537$	3	17.4 ± 2.78	$100,046 \pm 6,767$
400	17.8 ± 4.33	$105,624 \pm 4,442$	5	17.6 ± 4.51	$96,223 \pm 4,441$
800	13.0 ± 3.82	$99,575 \pm 5,319$	10	12.1 ± 1.97	$94,157 \pm 4,165$
1,600	12.6 ± 2.70	$102,449 \pm 3,654$	15	15.9 ± 6.15	$102,793 \pm 4,071$
3,200	14.2 ± 2.76	$109,711 \pm 1,460$	20	12.6 ± 2.70	$102,449 \pm 3,654$
6,400	15.7 ± 3.74	$110,860 \pm 2,058$	25	14.7 ± 4.24	$107,972 \pm 2,447$
12,800	19.0 ± 3.93	$110,866 \pm 2,488$			

$m = 20$

$n = 1600$

None of the direct-search methods were able to solve the multiplexer task within the allotted 300,000 steps. The unguided random search showed no improvement over time because the weight vectors being tested were not dependent on previous search steps. Its final performance level is slightly better than that of the single-layer system ($\nu = 17$ versus $\nu = 24$). The guided random search does show improvement over time, though its learning curve becomes approximately flat early in the runs. Averaged over the last 3,000 steps of every run, the number of errors per step is approximately 0.35. The polytope algorithm performs better than both random search methods, reaching an average over the last 3,000 steps of 0.28 errors per step.

Error Back-Propagation Methods

Next we discuss some error back-propagation methods for learning in hidden units, starting with a method studied by Rosenblatt [42].

Rosenblatt's Back-Propagation Method

Rosenblatt is known for his work with the perceptron-family of learning rule, but his error back-propagation method has received little attention. Since this was proposed early in the history of research on learning in multilayer systems and seemed to work reasonably well for the experiments Rosenblatt performed, we wished to include it in our study. Rosenblatt's back-propagation method is a nondeterministic way to assign errors to hidden units based on the errors of output units. The following is our specification of Rosenblatt's back-propagation method:

1. Initialize all weights to zero.
2. Receive input vector, calculate the output of all units using a linear threshold function, and receive error signals for the output units.
3. Apply the perceptron learning rule to the output units.
4. Calculate the error, δ_{jk} , passed back from output unit k to hidden unit j (probabilistically based on the output unit's error, the weight connecting unit j to unit k , and the output of unit j).

$v_k[t]$ random variable from a uniform probability density function over $[0, 1]$, where $k \in O$ takes the values of the indices of the output units,

$$\delta_{jk}[t] = \begin{cases} 1, & \text{if } y_j[t] = 1 \text{ and } (d_j[t] - y_j[t]) w_{jk}[t] < 0 \text{ and } v_k[t] < p_1; \\ +1, & \text{if } y_j[t] = 0 \text{ and } (d_j[t] - y_j[t]) w_{jk}[t] < 0 \text{ and } v_k[t] < p_2 \\ & \text{or} \\ & \text{if } y_j[t] = 0 \text{ and } (d_j[t] - y_j[t]) w_{jk}[t] < 0 \text{ and } v_k[t] < p_3; \\ 0, & \text{otherwise.} \end{cases}$$

5. Apply the perceptron learning rule to each hidden unit j , using the sign of the sum of the back-propagated errors from the output units as the error signals:

$$\Delta w_{ij}[t] = -\rho \operatorname{sgn} \left(\sum_{k \in O} \delta_{jk}[t] \right) x_i[t].$$

6. Repeat, starting at Step 2, until the prespecified number of time steps has elapsed.

Rosenblatt's back-propagation method depends on the parameter ρ , a factor determining the magnitude of change for each weight, and the parameters p_1 , p_2 , and p_3 , which are probabilities affecting the frequencies with which the back-propagated error signals take the values $+1$ and -1 . Rosenblatt performed a number of experiments and determined that the values $p_1 = 0.9$, $p_2 = 0.3$, and $p_3 = 0.1$ were reasonable values. Rather than attempting to optimize all four parameters, we used these values for p_1 , p_2 , and p_3 for all experiments, only varying the value of ρ .

The results are in Table 4. There are few significant differences for different values of ρ , although values from 0.125 to 0.5 resulted in slightly lower values of ν . The values of ν and μ show no improvement over a single-layer system. Indeed, the learning curve for Rosenblatt's method in Figs. 14 and 4 shows no improvement over time and is always worse than the single-layer level. The learning curve is averaged over 30 runs of 300,000 steps each, giving values of $\nu = 23.9 \pm 1.58$ and $\mu = 121,115 \pm 92$. To judge the performance of Rosenblatt's back-propagation method fairly, additional values of p_1 , p_2 , and p_3 must be tested.

Rumelhart, Hinton, and Williams

Another approach to the back-propagation of errors was taken by Rumelhart, Hinton, and Williams [44]. Our specialization of this method to the two-layer multiplexer network is as follows:

1. Randomly initialize all weights to be in the interval $[-0.1, 0.1]$.

Table 4: Rosenblatt's Back-Propagation Method on the Multiplexer Task

ρ	ν	μ
0.030	24.7 ± 3.54	$121,085 \pm 238$
0.060	23.6 ± 2.96	$121,286 \pm 168$
0.125	22.8 ± 3.11	$121,112 \pm 156$
0.250	22.0 ± 2.92	$121,129 \pm 176$
0.500	21.9 ± 2.83	$121,175 \pm 278$
1.000	24.0 ± 2.74	$121,161 \pm 169$
2.000	23.1 ± 3.18	$121,132 \pm 134$

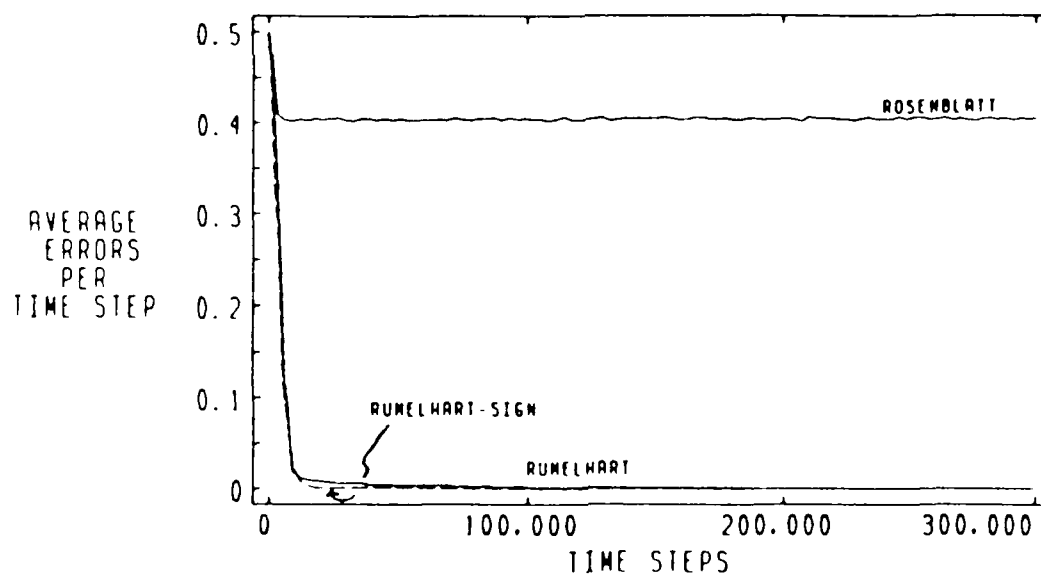


Figure 12: Learning Curves for Error Back-propagation Methods

2. Receive input vector, calculate output of all units, and receive error signals for the output units. All units use the semilinear output function:

$$y_j[t] = \frac{1}{1 + e^{-\sum_{i=0}^6 w_{ij}[t] x_i[t]}}.$$

3. Calculate δ_k for each output unit $k \in O$:

$$\delta_k[t] = (d'_k[t] - y_k[t]) y_k[t] (1 - y_k[t]),$$

where d'_k is a modified version of the desired output, defined as

$$d'_k[t] = \begin{cases} 0.9, & \text{if } d_k[t] = 1; \\ 0.1, & \text{if } d_k[t] = 0. \end{cases}$$

4. Apply the learning rule to the weights of each output unit k :

$$\Delta w_{jk}[t] = \rho \delta_k[t] x_j[t] + \rho_m \Delta w_{jk}[t - 1],$$

where $x_j[t]$ is an input component received by output unit k . Recall that the output units receive the original input terms to the system plus the output of the hidden units.

5. Calculate δ_j for each hidden unit j :

$$\delta_j[t] = \left(\sum_{k \in O} \delta_k[t] w_{jk}[t] \right) y_j[t] (1 - y_j[t]).$$

6. Apply the learning rule to the weights of each hidden unit j :

$$\Delta w_{ij}[t] = \rho \delta_j[t] x_i[t] + \rho_m \Delta w_{ij}[t - 1].$$

7. Repeat, starting with Step 2, until the prespecified number of time steps have elapsed.

By adding a fraction of the previous Δw to the current weight change, (Steps 4 and 6), it is hoped that the weight values will be more likely to follow the slope of the error function at the bottom of steep valleys, by canceling opposing steps up one side or the other. Rumelhart et al. consider this additional term as affecting the "momentum" of the trajectory of weight values. The method has two parameters: the rate of change parameter ρ and the factor ρ_m that controls the magnitude of the momentum term. Table 5 shows the values of ρ and ρ_m that were tested and the results averaged over 10 runs of 100,000 steps each.

Note the modification of the desired output value in Step 3. Rather than values of 1 and 0, values of 0.9 and 0.1 are used. Without this modification, weight values can grow in magnitude to the point where truncation errors due to the particular computer implementation can cause weight values to become frozen—the value of $y(1 - y)$ in the weight update equation becomes equal to zero.

Table 5: Rumelhart et al. Error Back-propagation Method on the Multiplexer Task

ρ	ν	μ
0.05	35188 ± 63	19.8 ± 0.84
0.10	31716 ± 1602	11.7 ± 3.30
0.25	14144 ± 1426	0.3 ± 0.55
0.50	6966 ± 1052	0.3 ± 0.39
1.00	4944 ± 1224	0.7 ± 0.39
2.00	3289 ± 935	0.2 ± 0.52
4.00	3294 ± 836	0.2 ± 0.52
8.00	13446 ± 4097	6.6 ± 2.93
16.00	32422 ± 5497	18.3 ± 3.12
$\rho_m = 0$		

ρ	ν	μ	ρ	ν	μ
0.05	34976 ± 496	18.9 ± 1.97	0.05	6130 ± 349	0.1 ± 0.26
0.10	33218 ± 1671	15.9 ± 4.57	0.10	3207 ± 454	0.0 ± 0.00
0.25	26245 ± 7354	9.3 ± 7.79	0.25	1747 ± 480	0.0 ± 0.00
0.50	11287 ± 2562	0.1 ± 0.26	0.50	1492 ± 844	0.2 ± 0.52
1.00	3836 ± 869	0.2 ± 0.52	1.00	5802 ± 2686	1.9 ± 1.86
2.00	3267 ± 1229	1.0 ± 0.86			
4.00	8905 ± 2213	3.5 ± 1.55			
$\rho_m = 0.5$					
$\rho_m = 0.9$					

The output value of a semilinear unit is a real value between 0 and 1. To compare with the other methods that use binary-valued, linear threshold units as the output unit, the output of the output unit k was set to 1 if $y_k \geq 0.5$ and was otherwise set to 0 while calculating μ and ν and the learning curve. This is only done in measuring performance, not in actually running the learning method.

From Table 5 one can see that this error back-propagation method reliably solved the multiplexer task within 100,000 steps, for $\rho = 0.1$ and 0.25 and $\rho_m = 0.9$. For $\rho = 0.25$ only 1,747 errors were accumulated over 100,000 steps ($\mu = 1,747$). Best performance (considering both ν and μ) resulted when $\rho = 0.25$ and $\rho_m = 0.9$. These parameter values were used to generate the learning curve shown in Fig. 4 and in Fig. 14, averaged over 30 runs of 300,000 steps each. The curve shows that extremely good performance is achieved very early in the runs; as early as 6,000 steps the average number of errors per step is below 0.06. The performance measures associated with this learning curve are $\nu = 0.00 \pm 0.00$ and $\mu = 1,962 \pm 148$.

The third curve in Fig. 4 shows the results of an experiment designed to test a modification to Rumelhart et al.'s method proposed by Sutton [48]. He suggested that it is the sign of the weight value appearing in the expression in Step 5 above that is the important contribution of the weight, and that the magnitude of it might hamper the method's progress, particularly when the magnitude is very small. We tested this hypothesis by replacing w_{jk} with the sign of w_{jk} , resulting in a new expression for $\delta_j[t]$:

$$\delta_j[t] = \left(\sum_{k \in O} \delta_k[t] \operatorname{sgn}(w_{jk}[t]) \right) y_j[t] (1 - y_j[t]).$$

As before, we varied ρ , with the results shown in Table 6 which are averaged over 10 runs of 100,000 steps each. The best value of ρ is still 0.25 and for ρ_m it is 0.9. The results averaged over 30 runs of 300,000 steps, using these parameter values, are $\nu = 0.00 \pm 0.00$ and $\mu = 1,354 \pm 575$, and the learning curve is shown in Fig. 4. The modification appears to retard the method's initial progress, but the task is still reliably solved. The cumulative error measure, μ , is not significantly different from that of the unmodified Rumelhart method. The modified method does appear to be

more robust than the unmodified method; the task is reliably solved ($\nu = 0.00$) for a wider range of parameter values.

Table 6: Sutton's Modification of the Error Back-propagation Method on the Multiplexer Task

ρ	ν	μ						
0.10	27759 ± 3438	7.2 ± 3.74						
0.25	11594 ± 958	0.1 ± 0.26						
0.50	5846 ± 1370	0.0 ± 0.00						
1.00	3013 ± 573	0.1 ± 0.26						
2.00	2336 ± 355	0.1 ± 0.26						
4.00	4378 ± 1179	1.0 ± 0.86						
$\rho_m = 0$								
ρ	ν	μ	ρ	ν	μ			
0.10	16447 ± 2504	1.6 ± 1.45	0.05	5091 ± 538	0.0 ± 0.00			
0.25	5427 ± 447	0.0 ± 0.00	0.10	2411 ± 310	0.0 ± 0.00			
0.50	2742 ± 369	0.0 ± 0.00	0.25	1310 ± 404	0.0 ± 0.00			
1.00	1536 ± 192	0.0 ± 0.00	0.50	1353 ± 796	0.2 ± 0.00			
2.00	2173 ± 767	0.1 ± 0.26	1.00	2968 ± 1309	0.7 ± 0.77			
4.00	8524 ± 1175	3.6 ± 0.96						
$\rho_m = 0.5$								
$\rho_m = 0.9$								

Associative Reinforcement Learning

Four associative reinforcement-learning methods were studied, two being variants of one of the others. Barto and colleagues have developed several associative reinforcement-learning methods [7,6,14,47]. Sutton [47] compared a number of these methods. For tasks most similar to those faced by hidden units in the networks applied to the multiplexer task, Sutton found that a particular learning rule, which we will call "associative search with reinforcement prediction," or AS-RP, rule performed better than others.

Associative Search with Reinforcement Prediction

The AS-RP method employs an additional unit that adjusts its weights, v , in order to match as closely as possible the value of the reinforcement expected for acting in the presence of each input vector. This provides the hidden units with a “reference” signal to which the current reinforcement can be compared to determine whether it is greater or less than the reinforcement usually received when given the current input vector. One can think of this extra unit as a predictor of the reinforcement to be received. The AS-RP method is defined as follows:

1. Initialize all weights to zero.
2. Receive input vector, calculate output of all units, and receive error signals for the output units. The output, y_j , of hidden unit j is given by:

$$y_j[t] = \begin{cases} 1, & \text{if } \sum_{i=0}^6 w_{ij}[t] x_i[t] + \eta_j[t] > 0; \\ 0, & \text{otherwise,} \end{cases}$$

where the $\eta_j[t]$ are sequences of random variables with density function

3. Apply the perceptron learning rule to the output units.
4. Calculate the reinforcement signal for the hidden units:

$$r[t] = 1 - \frac{1}{|O|} \sum_{j \in O} |d_j[t] - y_j[t]|.$$

where m is the number of output units. Since $m = 1$, $r[t] \in \{0, 1\}$.

5. Calculate the prediction of reinforcement, r_p , as follows:

$$r_p[t] = \sum_{i=1}^n v_i[t] x_i[t],$$

where n is the number of input components to the system and $v_i[t]$ is the predictor-unit's weight associated with input component $x_i[t]$.

6. Apply the associative search rule to hidden unit j :

$$\Delta w_{ij}[t] = \rho (r[t] - r_p[t]) (y_j[t] - \pi_j[t]) x_i[t],$$

where

$$\pi_j[t] = E \{y_j[t] | w_j[t]; x[t]\},$$

which is the expected value of the output y_j of unit j , given its current weight values and input. Since $y_j \in \{0, 1\}$, π_j is the probability that $y_j = 1$.

7. Update the predictor's weights.

$$\Delta v_i[t] = \rho_p (r[t] - r_p[t]) x_i[t].$$

8. Repeat, starting with Step 2, until the prespecified number of time steps have elapsed.

Two parameters control this method: the rate of change in modifying the hidden units' weights is ρ , and the rate of change in modifying the reinforcement predictor's weights is ρ_p . Five values of ρ were tried while ρ_p was set to one of three values. For each set of parameter values, 10 runs were made of 300,000 steps each.

The results in Table 7 show that the AS-RP method did not completely solve the multiplexer task, but for $\rho = 0.16$ and $\rho_p = 0.01$ the value of ν was about 2.8, meaning that after 300,000 steps an average of only 2.8 out of 64 input vectors resulted in an incorrect output. The performance of the AS-RP method over time is shown by its learning curve in Figure 13. The learning curve is averaged over 30 runs using $\rho = 0.16$ and $\rho_p = 0.01$, and resulted in $\nu = 3.36 \pm 1.98$ and $\mu = 48,754 \pm 8,662$. Better performance might be attainable by testing additional parameter values.

Another way to improve this method's performance is to include the output of the hidden units in the set of input components to the reinforcement-predictor unit. It may be impossible for a single unit to implement an accurate mapping from input vectors to reinforcement values, just as it is impossible for a single unit to implement a multiplexer function. This possibility was not tested.

Table 7: Associative Search with Reinforcement Prediction on the Multiplexer Task

ρ	ν	μ	ρ	ν	μ
0.01	23.7 ± 3.56	$121,756 \pm 131$	0.01	24.3 ± 3.51	$121,867 \pm 224$
0.04	11.6 ± 4.70	$95,602 \pm 8,825$	0.04	10.2 ± 5.29	$90,639 \pm 14,970$
0.16	2.8 ± 3.12	$42,149 \pm 16,354$	0.16	5.7 ± 3.11	$61,708 \pm 15,251$
0.64	4.6 ± 4.86	$46,453 \pm 21,080$	0.64	9.3 ± 4.13	$65,222 \pm 16,051$
1.28	12.1 ± 5.56	$75,454 \pm 20,789$	1.28	5.4 ± 4.18	$40,169 \pm 20,829$

$\rho_p = 0.01$

$\rho_p = 0.03$

ρ	ν	μ
0.01	24.6 ± 2.78	$121,436 \pm 270$
0.04	13.7 ± 4.61	$95,407 \pm 8,819$
0.16	3.5 ± 2.04	$48,320 \pm 13,120$
0.64	4.7 ± 3.87	$50,817 \pm 24,206$
1.28	13.2 ± 4.50	$79,975 \pm 21,263$

$\rho_p = 0.1$

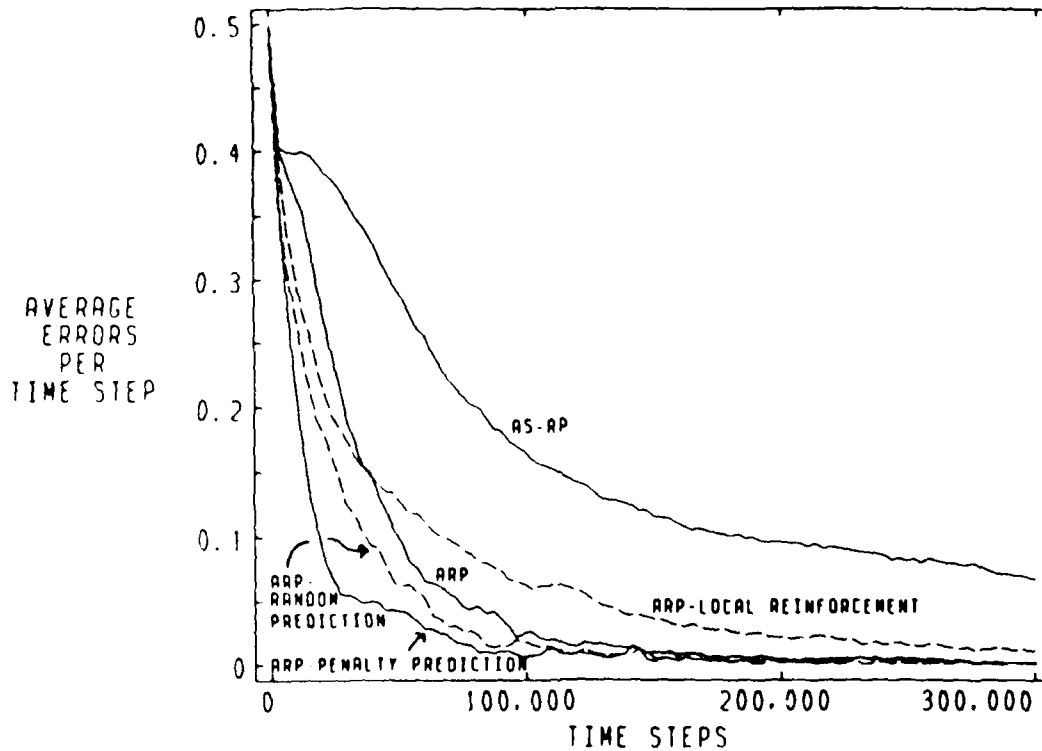


Figure 13: Learning Curve for Reinforcement Learning Methods on the Multiplexer Task

Associative Reward-Penalty

The second method from the reinforcement-learning class that we studied is the Associative Reward-Penalty, or A_{R-P} , learning rule described in Section 2.

1. Initialize all weights to zero.
2. Receive input vector, calculate output of all units, and receive error signals for the output units. Output functions are those used for the AS-RP method.
3. Apply the perceptron learning rule to the output units.
4. Calculate the global reinforcement signal for the hidden units.

$$r[t] = 1 - \frac{1}{|O|} \sum_{j \in O} |d_j[t] - y_j[t]|.$$

For the multiplexer task, $|O| = 1$, so $r[t] \in \{0, 1\}$, but in general $r[t] \in [0, 1]$.

5. Apply the A_{R-P} rule to each hidden unit.
6. Repeat, starting with Step 2, until the prespecified number of steps have elapsed.

The A_{R-P} method depends on two parameters. The rate of weight change is controlled by ρ and λ . If $\lambda = 0$, no change is made to the weight values when the "penalty" signal $r[t] = 0$ is received. See Section 2.

Table 8 contains the results of the A_{R-P} method on the multiplexer task, averaged over 10 runs of 300,000 steps each. Of the parameter values tested, $\rho = 1$ and $\lambda = 0.004$ resulted in the best performance, solving the task with a final number of errors over all input vectors of 0.02.

The learning curve in Fig. 13 shows that the A_{R-P} method performed much better than the AS-RP. Averaged over 30 runs of 300,000 steps each, and using $\rho = 1$ and $\lambda = 0.004$, the A_{R-P} rule resulted in $\nu = 0.01 \pm 0.01$ and $\mu = 15,725 \pm 3,129$. The value of 0.01 ± 0.01 for ν indicates that the solution to the multiplexer task was reliably found.

Table 8: A_{R-P} Method on the Multiplexer Task

λ	ν	μ	ρ	ν	μ
0.001	0.52 ± 1.30	$24,960 \pm 10,347$	0.1	1.30 ± 2.02	$51,109 \pm 10,157$
0.002	0.33 ± 0.80	$25,234 \pm 7,486$	0.2	0.08 ± 0.03	$25,377 \pm 5,479$
0.004	0.02 ± 0.01	$14,493 \pm 5,557$	0.4	1.80 ± 4.63	$20,301 \pm 7,050$
0.008	0.01 ± 0.01	$20,046 \pm 5,918$	0.8	0.71 ± 1.53	$17,320 \pm 7,553$
0.016	18.80 ± 6.65	$107,729 \pm 7,101$	1.0	0.02 ± 0.01	$14,493 \pm 5,557$
0.032	23.00 ± 3.26	$118,806 \pm 185$	1.6	0.01 ± 0.00	$15,167 \pm 3,908$
			3.2	11.60 ± 8.63	$81,798 \pm 16,013$

$$\rho = 1$$

$$\lambda = 0.004$$

Local Reinforcement

The AS-RP and the A_{R-P} methods function in a "global" reinforcement paradigm in which each hidden unit receives the same reinforcement signal. However, hidden units in a multilayer system can be provided with more informative evaluation information than that provided by the global reinforcement signal. We investigated one possible way of using this information to construct a unique "local" reinforcement signal to each hidden unit. The approach is similar to Rosenblatt's back-propagation method in its division into several cases according to units' outputs and weight values, but differs in that reinforcements are propagated rather than errors.

Steps 1 through 3 are identical to those of the A_{R-P} method.

4. Let $r_{jk}[t]$ be a reinforcement based on the output value of output unit k and the weight connecting hidden unit j to output unit k , defined as:

$$r_{jk}[t] = \begin{cases} 0.5, & \text{if } w_{jk}[t] = 0; \\ 1, & \text{if } w_{jk}[t] \neq 0 \text{ and } d_k[t] = y_k[t] \\ & \text{or} \\ & y_j[t] = 1 \text{ and } w_{jk}[t](d_k[t] - y_k[t]) > 0; \\ & \text{or} \\ & y_j[t] = 0 \text{ and } w_{jk}[t](d_k[t] - y_k[t]) < 0; \\ 0, & \text{otherwise.} \end{cases}$$

Calculate the local reinforcement signal for hidden unit j as:

$$r_j = \prod_{k \in O} r_{jk}[t],$$

though, for this task $O = \{5\}$, so $r_j = r_{j,5}$.

5. Apply the A_{R-P} rule to the hidden units, now using separate reinforcements for each:²

$$\Delta w_{ij}[t] = \rho r_j[t] (y_j[t] - \pi_j[t]) x_i[t] + \lambda \rho (1 - r_j[t]) (1 - y_j[t] - \pi_j[t]) x_i[t].$$

6. After the prespecified number of steps have elapsed the final-step performance measure ν is calculated.

The motivations for the cases in Step 4 is as follows. When a hidden unit has no influence on the output unit, i.e., $w_{jk} = 0$, then no preference in its output should be revealed. To accomplish this, r_{jk} is set to 0.5 regardless of the output of the hidden unit, the output unit, and the correct output. The second case is composed of three situations. First, if the hidden unit does have a nonzero output weight, i.e., $w_{jk} \neq 0$, and the output unit generated a correct response, then the hidden unit is "rewarded" by being assigned a reinforcement value of 1, increasing the probability of the output value that it just produced. The second part rewards the hidden unit if its output value is 1 and its output weight has the same sign as the output unit's error. The third part rewards the unit when its output value is 0 and its output weight differs in sign from the output unit's error.

This modification to the A_{R-P} rule does not add any new parameters. We tried a number of values for ρ and λ and averaged the results over 10 runs of 300,000 steps each. From Table 9 we see that $\rho = 0.5$ and $\lambda = 0.0001$ resulted in the best value of ν , which was 0.55 errors over the 64 input vectors after 300,000 steps. The cumulative measure, μ , was lowest for $\lambda = 0.0005$.

²This can be called the "S-model" A_{R-P} rule following the terminology used in the study of stochastic learning automata [36]. It is applicable if the reinforcement value, r , is a real number in the interval $[0, 1]$. Note that it specializes to the version of the A_{R-P} method given by Equation 2.4 if the reward and penalty values of r are respectively represented by 1 and 0.

Table 9: A_{R-P} with Local Reinforcement on the Multiplexer Task

λ	ν	μ	ρ	ν	μ
0.0	1.32 ± 2.24	$16,069 \pm 10,750$	0.01	23.72 ± 3.05	$12,2127 \pm 116$
0.00001	3.07 ± 2.84	$28,822 \pm 15,678$	0.25	2.84 ± 3.61	$40,581 \pm 16,852$
0.0001	0.24 ± 0.52	$12,512 \pm 4,937$	0.50	0.55 ± 1.27	$17,109 \pm 7,221$
0.0002	1.07 ± 1.72	$17,830 \pm 7,706$	0.75	1.31 ± 1.69	$23,290 \pm 10,555$
0.0005	0.39 ± 0.55	$10,418 \pm 1,973$	1.00	0.24 ± 0.52	$12,512 \pm 4,937$
0.001	0.76 ± 0.87	$14,539 \pm 1,997$	1.25	1.44 ± 2.66	$22,467 \pm 12,298$
0.002	4.64 ± 5.51	$21,145 \pm 2,279$			
0.004	6.53 ± 4.69	$34,506 \pm 1,613$			
0.008	10.30 ± 3.18	$64,436 \pm 1,644$			

 $\rho = 1$ $\lambda = 0.0001$

A learning curve for the A_{R-P} with local reinforcement, again averaged over 30 runs of 300,000 steps each, is included in Fig. 13. The values $\rho = 0.6$ and $\lambda = 0.0001$ were used for the method's parameters. This modification to the A_{R-P} method performs slightly better than the original A_{R-P} method before approximately the 20,000th step, and thereafter its performance is worse than that of the A_{R-P} .

The local reinforcement addition seems to help during the early stages, but is a hindrance throughout the remainder of a run. Perhaps this indicates that using the information about the hidden units' output weights and the output units' errors is only beneficial while the hidden units have minor effects on the output unit through output weights of small magnitudes. When output weights are near zero, learning according to the A_{R-P} method with global reinforcement is very slow because there is very little correlation between a hidden unit's output and the global reinforcement signal. But as the output weights increase in magnitude they acquire more of an influence on the global reinforcement and can begin to optimize their weight values. A more complex task—one requiring more than a single output unit—might demonstrate a greater potential for using this scheme for calculating local reinforcement.

Penalty Prediction

The plight of a hidden unit that has not yet acquired, or has lost, a substantial influence on the output units is to learn very slowly if it is modifying its weights through efforts to increase the reinforcement value. Is there some way to put such an "unused" unit to better use? We applied a second extension of the A_{R-P} method to the multiplexer task to investigate this question. This extension is based on the assumption that poor performance is caused by the lack of an appropriate representation. Situations for which incorrect outputs are generated need to be represented differently, perhaps with additional components, giving the output units more degrees of freedom with which they can alter their outputs.

To realize this idea we divide the learning rule for the hidden units into two parts, each part coming into play at different stages. When a hidden unit has a substantial effect on units "downstream," then the normal A_{R-P} learning rule is followed. But when a hidden unit does not significantly influence other units, the hidden unit adjusts its weights in an attempt to match them to input vectors that result in low reinforcement values, in effect becoming a "penalty predictor." In this way, new features are introduced that represent inputs for which the performance of the system is low. This is related to the data-directed method of Reilly, Cooper, and Elbaum [40], who dedicate new hidden units whenever an error is encountered by their system.

The implementation of this strategy depends on a measure of the degree to which a hidden unit has an influence on other units. Since in this task a hidden unit can only influence one other unit, we simply used the magnitude of the hidden unit's single output weight as an indication of influence, though, as Klopff and Gose [26] showed, other measures might lead to more accurate indicators of influence. The magnitude of a hidden unit's output weight is squashed into the range $[0, 1]$ by passing it through a logistic function. Some method of combining the measures from different output weights must be employed when the network has more than one output unit. The A_{R-P} method is modified as follows:

Steps 1 through 4 are identical to those of the A_{R-P} method.

5. Apply the A_{R-P} rule with penalty prediction to the hidden units:

(a) Calculate the influence, α_j , of hidden unit j on the output units:

$$\alpha_j[t] = \frac{1}{1 + e^{\frac{w_{jk}[t] - w_\alpha}{\tau_\alpha}}}.$$

(b) Update the weights:

$$\begin{aligned} \Delta w_{ij}[t] = & \alpha_j[t] \left(\begin{aligned} & \rho r[t] (y_j[t] - \pi_j[t]) x_i[t] \\ & + \lambda \rho (1 - r[t]) (1 - y_j[t] - \pi_j[t]) x_i[t] \end{aligned} \right) \\ & + (1 - \alpha_j[t]) \rho_\alpha (1 - r[t] - \pi_j[t]) x_i[t]. \end{aligned}$$

6. Same as Step 6 of the A_{R-P} method.

The equation in Step 5b is composed of two main parts. The first part is the expression for the S version of the A_{R-P} rule. Its contribution to the update of weights varies inversely with that of the second part of the equation. The second part is only significant when α_j is small, meaning that hidden unit j has little influence on the output unit. It serves to push the weight vector in the direction of the current input vector when r is small and it pushes the weight vector away from the input vector when r is large.

In addition to the parameters ρ and λ of the A_{R-P} method, this modification depends on the values of ρ_α , w_α , and τ_α , which have their strongest effect when α_j , the influence on the output units, is small. The variable α_j is a function of unit j 's output weights, defined in such a way as to scale its value between 0 and 1; $\alpha_j = 1$ when unit j has a very strong influence on an output unit, and $\alpha_j = 0$ when it has no influence. The scaling function for α is controlled by the parameters w_α and τ_α , and its form is that of the logistic function, where τ_α is the "spread" of the function and w_α is the value of its argument such that $\alpha_j = 0.5$ when $w_\alpha = w_{jk}[t]$. For example, if $w_\alpha = 1.5$ and $\tau_\alpha = 0.1$ and there is one output unit, then α_j will have the following values for the given values of unit j 's output weight:

output weight (w_{jk})	α_j
0	0.000
± 1	0.007
± 2	0.993
± 3	1.000

For the multiplexer task, the output unit learns under the perceptron learning constant with a learning constant of $\rho = 1$. Therefore, the output unit's weights, which are the output weights of the hidden units, will always be integer-valued. For $w_\alpha = 1.5$ and $\tau_\alpha = 0.1$, the A_{R-P} with penalty prediction method will approximate the original A_{R-P} method except when the value of the output weight is 0, as it is initially, or 1.

Table 10 shows the results of testing this method for various parameter values over 10 runs of 50,000 steps each. Using the values $\rho = 1$ and $\lambda = 0.004$, which gave the best performance for the original A_{R-P} method, we found that $\rho_\alpha = 16$, $w_\alpha = 1.5$, and $\tau_\alpha = 0.1$ were the best parameter values tested. Not reported here are further experiments in which ρ and λ are varied, again finding $\rho = 1$ and $\lambda = 0.004$ to be the best of a small set of alternative values.

The best parameters were used to generate the learning curve in Fig. 13, averaged over 30 runs of 300,000 steps each. The learning curve shows that this method performed much better than the original A_{R-P} . The A_{R-P} with penalty prediction resulted in performance measures of $\nu = 0.08 \pm 0.17$ and $\mu = 7,411 \pm 1,773$. Roughly twice as many errors on average were made during the runs of the A_{R-P} method ($\mu = 7,411$ versus $\mu = 15,725$).

The fact that large values of ρ_α result in better performance than small values suggests that the advantage of the penalty prediction modification is due to the size of the large jumps in a hidden unit's weights when the unit has a small output weight, and not in the direction of the weight change. This hypothesis was tested through further experiments, as follows. The method was modified in a way that preserved the size of the large weight changes while removing the dependence on the reinforcement signal to direct the weight change. Instead, a random signal guided the changes in weight values when the output weight is of low magnitude. Thus, Step 5b of the A_{R-P} with penalty prediction becomes

$$\Delta w_{ij}[t] = \alpha_j[t] \left(\begin{array}{l} \rho r[t] (y_j[t] - \pi_j[t]) x_i[t] \\ + \lambda \rho (1 - r[t]) (1 - y_j[t] - \pi_j[t]) x_i[t] \end{array} \right) + (1 - \alpha_j[t]) \rho_\alpha (s_j[t] - \pi_j[t]) x_i[t],$$

where the $s_j[t]$ are sequences of Bernoulli random variables (possible values are 0 and

Table 10: A_{R-P} with Penalty Prediction on the Multiplexer Task

ρ_{it}	w_{it}	ν	μ	τ_{it}	ν	μ
0	0.5	7.30 ± 5.06	$11,246 \pm 1,965$	0.01	2.78 ± 2.66	$7,695 \pm 2,304$
1	0.5	23.50 ± 3.11	$20,009 \pm 223$	0.1	0.26 ± 0.52	$4,761 \pm 2,225$
2	0.5	19.90 ± 6.85	$18,004 \pm 2,032$	0.2	0.48 ± 1.05	$8,698 \pm 3,030$
4	0.5	4.70 ± 5.51	$10,795 \pm 3,324$	0.4	18.00 ± 5.89	$18,542 \pm 2,108$
8	0.5	0.54 ± 1.21	$4,682 \pm 1,498$	0.6	23.80 ± 2.90	$20,242 \pm 100$
16	0.5	3.20 ± 3.06	$6,143 \pm 2,389$	0.8	23.20 ± 2.27	$20,340 \pm 64$
32	0.5	7.10 ± 5.29	$7,712 \pm 2,734$	1.0	23.70 ± 2.13	$20,243 \pm 74$
4	1.0	22.60 ± 3.35	$19,786 \pm 325$	$\rho = 1.0, \lambda = 0.004$ $\rho_{it} = 0.1, w_{it} = 1.5$		
8	1.0	6.20 ± 6.75	$10,142 \pm 4,136$			
16	1.0	0.37 ± 0.52	$6,779 \pm 1,626$			
32	1.0	3.80 ± 3.85	$8,253 \pm 1,723$			
4	1.5	14.70 ± 7.79	$17,297 \pm 2,781$	$\rho = 1.0, \lambda = 0.004$ $\tau_{it} = 0.1$		
8	1.5	4.20 ± 5.51	$8,301 \pm 2,403$			
16	1.5	0.26 ± 0.52	$4,761 \pm 2,225$			
32	1.5	5.20 ± 5.52	$8,140 \pm 3,714$			
4	2.0	25.00 ± 3.66	$20,209 \pm 53$	$\rho = 1.0, \lambda = 0.004$ $\tau_{it} = 0.1$		
8	2.0	19.40 ± 6.28	$19,185 \pm 1,477$			
16	2.0	12.70 ± 6.68	$14,733 \pm 4,688$			
32	2.0	4.00 ± 3.84	$10,030 \pm 3,328$			
4	4.0	23.80 ± 1.32	$20,234 \pm 55$	$\rho = 1.0, \lambda = 0.004$ $\tau_{it} = 0.1$		
8	4.0	22.10 ± 3.19	$20,252 \pm 66$			
16	4.0	23.70 ± 3.12	$20,254 \pm 66$			
32	4.0	23.00 ± 3.21	$20,240 \pm 72$			

$\rho = 1.0, \lambda = 0.004$
 $\tau_{it} = 0.1$

1).

The learning curve for the A_{R-P} with random prediction is shown in Fig. 13. Its performance is worse than that of the A_{R-P} with penalty-prediction method, suggesting that there is an advantage in predicting penalties. However, it performs better than the simple A_{R-P} method. Thus, there is also an advantage to taking undirected, large steps in the search for weight values for unused units. The increase in performance of the A_{R-P} with penalty prediction method over the simple A_{R-P} is probably due to both effects.

Summary of Comparative Simulations

To facilitate the comparison of the learning methods' performance on the multiplexer task, most of the learning curves are superimposed in Fig. 14. Recall that the errors per time step are plotted by averaging over 30 runs and over bins of 3,000 step intervals. A non-learning, random strategy of selecting outputs would result in an average of 0.5 errors per time step.

It is easily seen that the classes of methods in order of decreasing performance are

1. error back-propagation (excluding Rosenblatt's method),
2. reinforcement learning, and
3. direct search.

This ranking is supported by the values of the performance measures, shown in Table 11, where the methods are ranked according to their resulting values of μ . There is no statistically-significant difference between the values of μ for the two versions of the Rumelhart et al. method. However, the difference between these methods and the best reinforcement-learning method, the A_{R-P} with penalty prediction, is significant.

Among the reinforcement-learning methods, some differences in μ are significant, while others are not. In particular, the results of the AS-RP method are significantly

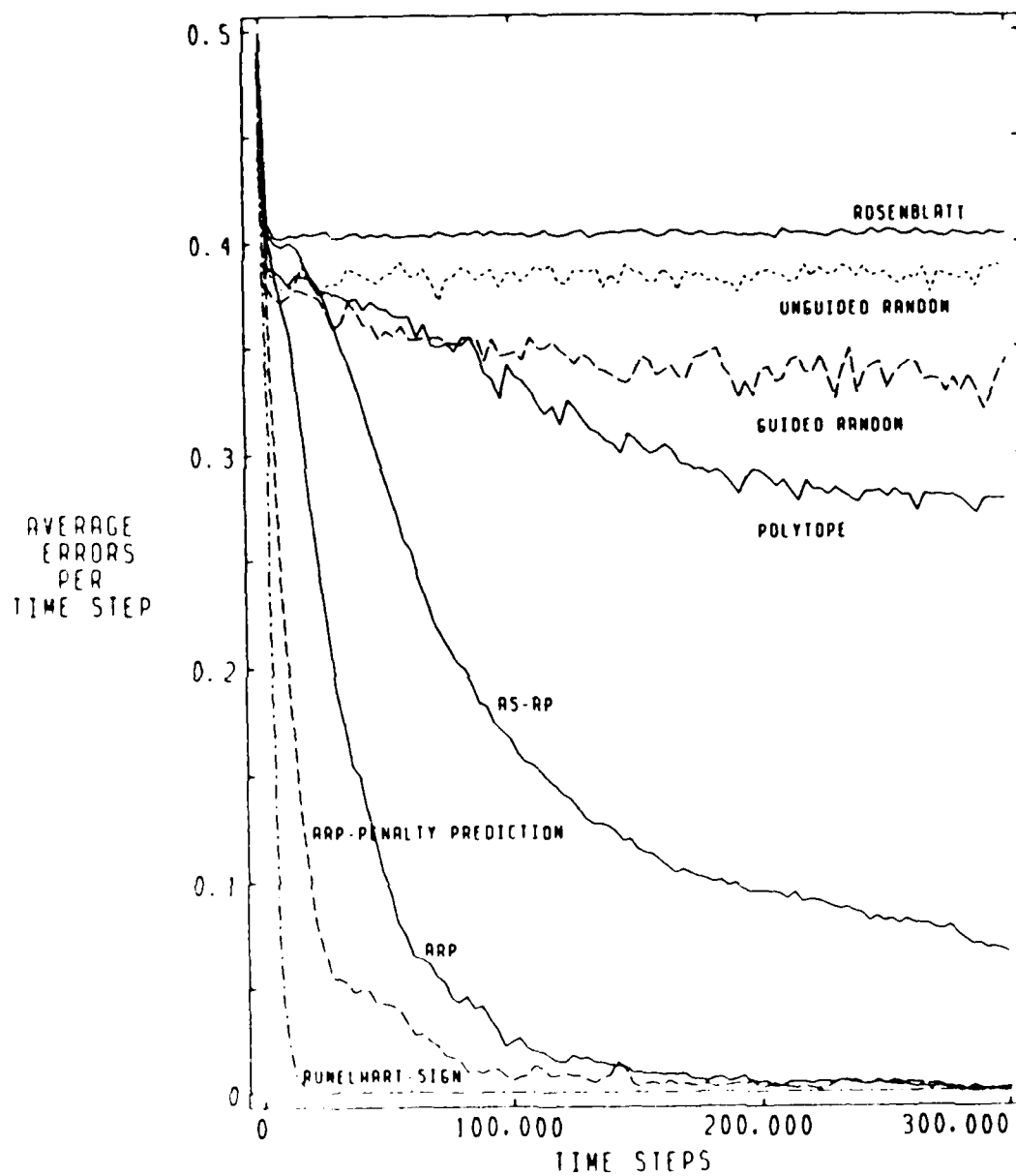


Figure 14: Learning Curves for All Methods on the Multiplexer Task

Table 11: Performance Summary for Multiplexer Task

method	ν	μ	parameters
Rumelhart sign of output weight	0.00 ± 0.00	$1,354 \pm 575$	$\rho = 0.25, \rho_m = 0.9$
Rumelhart	0.00 ± 0.00	$1,962 \pm 148$	$\rho = 0.25, \rho_m = 0.9$
A _{R-P} with penalty prediction	0.08 ± 0.17	$7,411 \pm 1,773$	$\rho = 1, \lambda = 0.004,$ $\rho_\alpha = 16, w_\alpha = 1.5, \tau_\alpha = 0.1$
A _{R-P} with random prediction	0.01 ± 0.00	$10,695 \pm 2,690$	$\rho = 1, \lambda = 0.004,$ $\rho_\alpha = 16, w_\alpha = 1.5, \tau_\alpha = 0.1$
A _{R-P}	0.01 ± 0.01	$15,725 \pm 3,129$	$\rho = 1, \lambda = 0.004$
A _{R-P} with local reinforcement	0.65 ± 1.08	$20,467 \pm 6,923$	$\rho = 0.6, \lambda = 0.0001$
AS-RP	3.36 ± 1.98	$48,754 \pm 8,662$	$\rho = 0.16, \rho_p = 0.01$
polytope	14.2 ± 2.09	$94,977 \pm 3,079$	$n = 1600, m = 10,$ $c_r = 2, c_e = 2, c_c = 0.2$
guided random	13.1 ± 2.36	$103,866 \pm 3,420$	$n = 3200, \tau = 1$
unguided random	17.0 ± 2.93	$115,062 \pm 229$	$n = 1600$
Rosenblatt	23.9 ± 1.58	$121,115 \pm 92$	$\rho = 0.5,$ $p_1 = 0.9, p_2 = 0.3, p_3 = 0.1$

worse than all other reinforcement-learning results. As discussed earlier, another version of the AS-RP method should be tested: the output of the hidden units should be included as input to the reinforcement predictor, thus not restricting the reinforcement prediction to be a linear function of the input as originally represented. All other differences are significant. The direct search methods are significantly worse than others and their relative ranking is also significant.

Now we can ask what new features actually developed during successful runs of multilayer learning methods, that is, to what sets of input patterns did the hidden units tune? For a partial answer to this question, we analyzed two runs, one with Rumelhart et al.'s method and the other with the A_{R-P} with penalty prediction method. Each run was interrupted at three points to determine the features that the hidden units had acquired at various stages. Fig. 14 shows that a single run using the Rumelhart et al. method is very likely to have solved the multiplexer task by the 10,000th step, so the run was analyzed after 2,000, 5,000, and 10,000 steps. The results of this analysis appear in Table 12. A unit's state is specified by a logical expression for the union of all input vectors for which the output of the unit is 1. For example, unit 1 on the 2,000th step responds with output 1 for input vectors $(0, 0, 0, 0, 0, 1)^T$ and $(0, 0, 0, 1, 0, 1)^T$ (disregarding the constant component of the input vectors). Labeling the components of the input vectors as $(a_1, a_2, d_1, d_2, d_3, d_4)$, for address lines a_1, a_2 and data lines d_1, d_2, d_3, d_4 . A minimal logical expression for the union of these vectors is $\bar{a}_1 \bar{a}_2 \bar{d}_1 \bar{d}_3 d_4$. Included with each hidden unit expression is the approximate value of the unit's output weight, indicating how that unit affects the activation of the output unit.

In addition to the hidden-unit analysis, expressions were determined for the output unit, both with and without the features generated by the hidden units. Let us start our discussion of Table 12 with these expressions, by first studying the last row. At step 2,000, a relatively complex expression developed for the output unit, but by step 10,000 the unit's expression is exactly the multiplexer expression, as expected. The expressions for the output unit without hidden units show that at step 10,000 the new features learned by the hidden units are necessary for the generation of the correct output for input vectors containing three of the four possible addresses; for

Table 12: New Features Developed by the Error Back-Propagation Method

	Step 2,000	Step 5,000	Step 10,000
Unit 1	$\bar{a}_1 \bar{a}_2 \bar{d}_1 \bar{d}_3 d_4$ (-2)	$\bar{a}_1 \bar{a}_2 (\bar{d}_1 \bar{d}_3 \vee \bar{d}_1 d_2 d_4) \vee$ $a_1 \bar{a}_2 (\bar{d}_1 d_2 \bar{d}_3 \vee \bar{d}_1 \bar{d}_3 d_4)$ (-7)	$\bar{a}_1 \bar{a}_2 (\bar{d}_1 \bar{d}_3 \vee \bar{d}_1 d_4) \vee$ $a_1 \bar{a}_2 (\bar{d}_1 \bar{d}_3 \vee \bar{d}_1 d_4)$ (-7)
Unit 2	$\bar{a}_1 \bar{a}_2 \bar{d}_1 \bar{d}_3 d_4$ (-2)	$\bar{a}_1 \bar{a}_2 \bar{d}_1$ (-7)	$\bar{a}_1 \bar{a}_2 \bar{d}_1$ (-11)
Unit 3	null (-1)	$\bar{a}_1 \bar{a}_2 \bar{d}_2 d_3 d_4 \vee$ $\bar{a}_1 a_2 (\bar{d}_2 d_4 \vee \bar{d}_2 d_3 \vee \bar{d}_1 \bar{d}_2)$ (-9)	$\bar{a}_1 a_2 \bar{d}_2$ (-12)
Unit 4	null (-1)	$\bar{a}_1 \bar{a}_2 (\bar{d}_1 \bar{d}_3 d_4 \vee d_2 \bar{d}_3 d_4) \vee$ $a_1 \bar{a}_2 (\bar{d}_1 \bar{d}_3 d_4 \vee d_2 \bar{d}_3 d_4)$ (-3)	$a_1 \bar{a}_2 \bar{d}_3$ (-7)
output unit without hidden units	$\bar{a}_1 \bar{a}_2 (d_3 d_4 \vee d_2 d_3$ $\vee d_2 d_4 \vee d_1 \bar{d}_2 d_4$ $\vee d_1 d_3 \bar{d}_4 \vee d_1 d_2 \bar{d}_3) \vee$ $\bar{a}_1 a_2 (d_1 d_2 d_4 \vee d_1 d_3 d_4$ $\vee d_1 d_2 d_3 \vee d_2 d_3 d_4) \vee$ $a_1 \bar{a}_2 (d_3 d_4 \vee d_2 d_3$ $\vee d_2 d_4) \vee$ $a_1 a_2 (d_1 d_2 d_4 \vee d_1 d_3 d_4$ $\vee d_1 d_2 d_3 \vee d_2 d_3 d_4)$	$\bar{a}_1 \bar{a}_2 \vee$ $\bar{a}_1 a_2 \vee$ $a_1 \bar{a}_2 \vee$ $a_1 a_2 (d_4 \vee d_2 d_3)$	$\bar{a}_1 \bar{a}_2 \vee$ $\bar{a}_1 a_2 \vee$ $a_1 \bar{a}_2 \vee$ $a_1 a_2 d_4$
output unit with hidden units	$\bar{a}_1 \bar{a}_2 (d_1 d_2 d_4 \vee d_1 d_3 d_4$ $\vee d_2 d_3) \vee$ $\bar{a}_1 a_2 (d_1 d_2 d_4 \vee d_1 d_3 d_4$ $\vee d_2 d_3) \vee$ $a_1 \bar{a}_2 (d_1 d_2 d_4 \vee d_1 d_3 d_4$ $\vee d_2 d_3) \vee$ $a_1 a_2 (d_1 d_2 d_4 \vee d_1 d_3 d_4$ $\vee d_1 d_2 d_3 \vee d_2 d_3 d_4)$	$\bar{a}_1 \bar{a}_2 d_1 \vee$ $\bar{a}_1 a_2 d_2 \vee$ $a_1 \bar{a}_2 (d_3 \vee d_1 d_2 d_3) \vee$ $a_1 a_2 (d_4 \vee d_2 d_3)$	$\bar{a}_1 \bar{a}_2 d_1 \vee$ $\bar{a}_1 a_2 d_2 \vee$ $a_1 \bar{a}_2 d_3 \vee$ $a_1 a_2 d_4$

address (1,1) the output unit itself is capable of producing the correct output.

Given the expression for the output unit without hidden units at step 10,000, it is clear how the terms formed by units 2, 3, and 4 are being used. All have negative influences on the output unit, effectively carving out of the output unit's expression those input vectors for which the output unit produces a 1 when the correct output is 0. The role played by unit 1 is much less clear, and would require a careful analysis of exact weight values for us to understand.

Table 13 shows the results of a similar analysis of a run with the A_{R-P} with penalty prediction method. The run was interrupted at 10,000, 20,000, and 50,000 steps, a larger total number of steps than was used for the analysis of the Rumelhart et al. method. The expression for the output unit with hidden units at the 50,000th step is indeed the multiplexer expression.

The manner in which the hidden units interact with the output unit to produce the correct output is not as straightforward as it was for the previous example. It is clear that unit 4 has acquired the same role here as it did in the other run. This is purely a matter of coincidence, since there is no a priori bias among the hidden units; initially, each unit is equally likely to develop a particular feature.

In conclusion, this section presents the results of several methods for learning in multilayer networks on a particular learning task. The task is of sufficient scale that direct search methods perform poorly, never solving the task within the allotted time. Several error back-propagation methods were studied, of which Rumelhart et al.'s method readily dealt with the difficulties of the task, reliably solving it within several thousand steps. Reinforcement-learning methods were also tested, with various degrees of success.

In comparing the performance results described in this section, it is important to keep in mind two critical limitations of this study. The most obvious limitation is that a single task was used. The results provide no indication of how the relative ranking of the methods would change if different tasks, either simpler or more complex, are used. Answers to the question of how well the methods scale-up to harder tasks require further experiments on tasks of varying complexity. A related issue is how a method's performance is affected by altering the network architecture, such as the

Table 13: New Features Developed by A_{R-P} with Penalty Prediction Method

	Step 10,000	Step 20,000	Step 50,000
Unit 1	$\bar{a}_1\bar{a}_2d_1\bar{d}_2\bar{d}_4\vee$ $a_1\bar{a}_2d_1\bar{d}_2d_3\bar{d}_4\vee$ $a_1a_2(d_1\bar{d}_2\bar{d}_4$ $\vee d_1\bar{d}_2d_3\bar{d}_4)$ (0)	null (-1)	$\bar{a}_1\bar{a}_2\bar{d}_1d_2\bar{d}_4$ (-7)
Unit 2	$\bar{a}_1\bar{a}_2(\bar{d}_1\bar{d}_2d_3\vee\bar{d}_1d_2)\vee$ $a_1\bar{a}_2\bar{d}_1d_2$ (-7)	$\bar{a}_1\bar{a}_2\bar{d}_1\vee$ $a_1\bar{a}_2\bar{d}_1d_2$ (-13)	$\bar{a}_1\bar{a}_2\bar{d}_1\vee$ $a_1\bar{a}_2\bar{d}_1d_2$ (-13)
Unit 3	$\bar{a}_1a_2d_4\vee$ $\bar{a}_1a_2\bar{d}_2d_4\vee$ $a_1\bar{a}_2\bar{d}_2d_4$ (-8)	$\bar{a}_1\bar{a}_2d_4\vee$ $\bar{a}_1a_2\bar{d}_2d_4\vee$ $a_1\bar{a}_2(\bar{d}_2d_4\vee d_2\bar{d}_3d_4)$ (-10)	$\bar{a}_1\bar{a}_2d_4\vee$ $\bar{a}_1a_2\bar{d}_2d_4\vee$ $a_1\bar{a}_2\bar{d}_2d_4$ (-16)
Unit 4	null (-1)	$a_1\bar{a}_2\bar{d}_3$ (-9)	$a_1\bar{a}_2\bar{d}_3$ (-25)
output unit without hidden units	$\bar{a}_1\bar{a}_2\vee$ $\bar{a}_1a_2(d_2d_3\vee d_4)\vee$ $a_1\bar{a}_2(d_4\vee d_3\bar{d}_4)\vee$ $a_1a_2d_4$	$\bar{a}_1\bar{a}_2\vee$ $\bar{a}_1a_2d_4\vee$ $a_1\bar{a}_2\vee$ $a_1a_2d_4$	$\bar{a}_1\bar{a}_2\vee$ $\bar{a}_1a_2(d_3\vee\bar{d}_2d_4)\vee$ $a_1\bar{a}_2\vee$ $a_1a_2d_4$
output unit with hidden unit	$\bar{a}_1\bar{a}_2(d_1\vee d_1\bar{d}_2\bar{d}_3)\vee$ $a_1a_2(d_2d_3\vee d_2d_4)\vee$ $a_1\bar{a}_2(\bar{d}_2d_3\vee d_2d_3d_4$ $\vee d_1d_2d_4\vee d_1d_2d_3)$	$\bar{a}_1\bar{a}_2d_1\vee$ $\bar{a}_1a_2d_2d_4\vee$ $a_1\bar{a}_2d_3\vee$ $a_1a_2d_4$	$\bar{a}_1\bar{a}_2d_1\vee$ $\bar{a}_1a_2d_2\vee$ $a_1\bar{a}_2d_3\vee$ $a_1a_2d_4$

addition or removal of hidden units. Neither issue was investigated by this study.

The second limitation is due to the manner in which the values for each method's parameters were chosen. The experimenter became part of every learning method by trying a number of different parameter values. The values that resulted in the best performance for a particular method were used in its comparison with the other methods. The time required to perform this parameter optimization process is not taken into account by the performance measures used in this study. A method might rank very well according to the performance measures but be very sensitive to its parameter values and require much effort to find optimal parameter values. This does not appear to be the case for the results reported in this section. The method with the best performance is Rumelhart's error back-propagation method modified to use the sign of the output weight, and it reliably solves the multiplexer task for a wide range of parameter values.

Some Further Experiments—The Batched A_{R-P} Method

In Section 3, I discussed the relationship between the operation of A_{R-P} networks and the error back-propagation scheme of Rumelhart et al. [44] and mentioned the result of Williams [61] that the weights in an A_{R-P} network (with $\lambda = 0$) move according to an unbiased estimate of the gradient of the global network reward probability. This fact suggests that it might be worthwhile to consider a sampling process in conjunction with the A_{R-P} method. If the units could obtain a better estimate of a true gradient through repeated sampling, then the performance would improve and in the limit approach the performance obtained with deterministic gradient descent methods. Furthermore, the network would retain its simple character, in that all units would still receive the same scalar signal.

To investigate this possibility, we considered a modification of the standard A_{R-P} learning procedure. The standard procedure consists of the following sequence of events which occurs each time a stimulus is presented: The network determines its output, this output is evaluated, the evaluation is broadcast to all units, and the units change their weights. The modification consists simply in allowing this updat-

ing sequence to take place several times during the presentation of a single stimulus. Furthermore, the weight changes induced by these updates are accumulated in a temporary location; only at the end of the stimulus presentation are the accumulated weight changes added to the actual weights. Geometrically, this procedure amounts to obtaining several sample vectors at a given point in weight space, and taking a step which is the resultant of the sample vectors. In the experiment to be reported below, the network computed a batch of ten such sample vectors for each stimulus presentation. We call this procedure the "batched" A_{R-P} method.

We wished to compare the batched procedure to the standard procedure in terms of the time needed to learn the multiplexer task. Note that the learning time is a function both of the direction and the size of the steps in weight space taken by the network. Since we were interested in the ability of the batched process to improve the direction of these steps, it was important to control for the step size. To do this, we first computed the average step size taken on the first few learning trials by networks using both the standard A_{R-P} learning procedure and the batched procedure.³ The ratio of these average step sizes was then used to scale the learning rates. In particular, the learning rate ρ for the standard procedure was chosen to be 0.5, and the learning rate for the batched procedure was then taken to be 0.079, so that the step size per stimulus presentation was the same in the two cases. Note that in the case of a deterministic method, the learning rate for the batched procedure would have to be 0.05, given that there are 10 samples per presentation; the actual step taken per stimulus presentation would be the same for the two procedures. For the stochastic method, however, the steps can be in different directions. The fact that the learning rate for the sampling procedure was larger than 0.05 indicates that the sample vectors tend to point in different directions and cancel, which is of course necessary if sampling is to have any effect.

The architecture used in this experiment was the same as that used in previous studies of the multiplexer--two layers of weights, with six input lines, four hidden units, and a single output unit (Fig. 10). The hidden units learned using the A_{R-P} rule, while the output unit learned using the perceptron rule. The evaluation

³The step size was computed as the Euclidean norm of the vector Δw .

signal was a deterministic function of the output of the network—if the output was correct, the evaluation was one, otherwise it was zero.

The results are shown in Fig. 15. The abscissa represents bins of 200 trials, where a trial refers to a single stimulus presentation. As discussed above, with this definition of a trial, the two learning procedures are equated in terms of average step size in weight space. The ordinate shows the average percentage error for passes through the 64 possible stimuli. This error is a mean over the 200 trials in the bins on the abscissa. Furthermore, each curve represents an average over 25 replications of the experiment. As can be seen, there is a substantial improvement in using the sampling procedure as compared to the standard procedure. If we use a percentage error of five percent as a learning criterion, then the sampling procedure learns 2.8 times faster than the standard procedure.

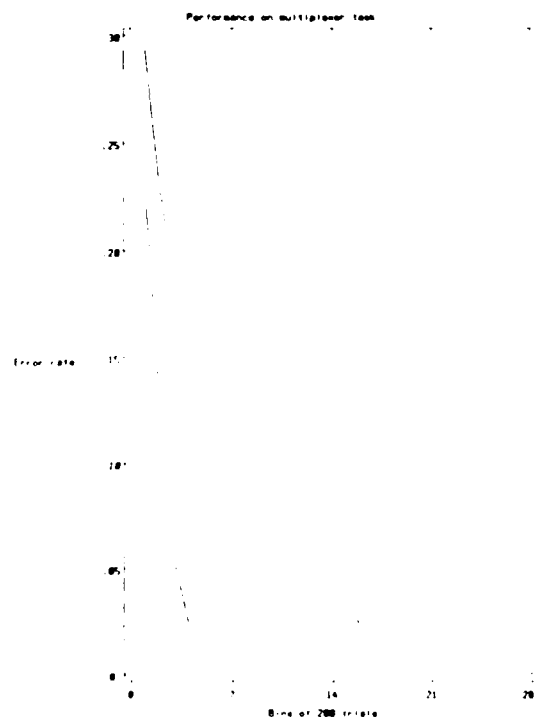


Figure 15: Learning Curves Comparing the Standard and Batched A_{R-P} Methods

This result shows that it is possible to obtain increasingly accurate estimates of a gradient, without requiring a complex error propagation process. There are both practical and biological implications of this result. Suppose, for example, that in some learning domain it is costly to obtain stimulus items, but that it is not costly to update the network and obtain evaluations. In such a domain, it might be practical to use the sampling procedure to speed learning. From a biological point of view, the batched approach emphasizes the point that the agent evaluating the output of a network need only be external to the network, and not necessarily external to the organism. If some internal agent has sufficient knowledge to be able to evaluate actions, in particular if the agent constitutes a model of the environment, then it is possible to improve learning through the batched method without going through the environment.

SECTION 5

POLE-BALANCING AGAIN

In previous research we used a version of the pole-balancing task (or inverted-pendulum task) to investigate the capabilities of the learning methods we had developed [13,47,15]. The pole-balancing task is an example of what can be called a strategy-learning task. A difficult temporal credit-assignment problem complicates this kind of learning—there is no standard with which to compare the system's actions on every step. This problem rules out the use of most connectionist learning methods for strategy learning because most of these methods require knowledge of the correct, or desired, actions for a training set of input vectors. Learning proceeds by the presentation of input vectors from the training set and the modification of weights in a manner that is dependent on the error between the correct action and the actual action, as was done for the experiments of Sections 3 and 4. For both the pole-balancing task and the Tower of Hanoi task described in Section 6, the training information arrives in the form of a failure or success signal after a series of actions.

Our earlier work with the pole-balancing task assumed the existence of a representation for the cart-pole system's state consisting of a large number of non-overlapping "boxes" produced by a pre-existing decoder. Given this representation, the task became one of filling in look-up tables—one to specify an evaluation function and one to specify control actions. This simplified representation allowed us to separate representation issues from the issues of temporal credit assignment. In the studies reported here, the pre-existing decoder is replaced by a layered adaptive network. This network receives as input a vector of four real numbers giving the state of the cart-pole system. The network has to learn how to represent the state so that the system as a whole can successfully avoid failure. The layered network provides a kind of adaptive decoder. In order to accomplish this, the Adaptive Critic Element

(ACE) and the Associative Search Element (ASE) of previous studies were combined with the error back-propagation method of Rumelhart et al. [44]. Consequently, the architecture used consists of two networks: the *evaluation network* for learning an evaluation function, which is an elaboration of the ACE, and the *action network* for learning action heuristics, which is an elaboration of the ASE.

The networks and learning methods are described first. Since the networks and the learning methods used in the pole-balancing task and the Tower of Hanoi puzzle (described in Section 6) are very similar, in this section we describe strategy learning networks in a way that is general enough to encompass the systems used in both of these tasks. We then briefly describe the pole-balancing simulation and how the learning networks interact with it. Finally, results of simulation experiments are described. This section is an edited form of Chapters V and VI of C. W. Anderson's dissertation [4].

Strategy Learning Networks

As presented, the networks have just two layers, but the learning methods are easily extended to additional layers. The evaluation network and action network do not necessarily have the same number of hidden or output units, but since the particular network being discussed is always obvious from the context, the same variables are used to index units in both networks. Let there be m_h hidden units and m_o output units, for a total of $m = m_h + m_o$ units. The hidden units are indexed from 1 to m_h and the output units are indexed from $m_h + 1$ to m . (The evaluation network has only a single output unit.) Let H and O respectively denote the sets of hidden units and the output units. The evaluation and action networks do not share hidden units, but since we wanted to avoid the difficulties of integrating the hidden-unit learning methods for the evaluation network and the action network,

the architecture is shown in Fig. 16. The triangles represent the "computation-places" and the input vectors arrive from the left and "pass through" their respective intersection points, represented by intersections of horizontal and vertical lines, to the right, where an output is computed and sent out the output

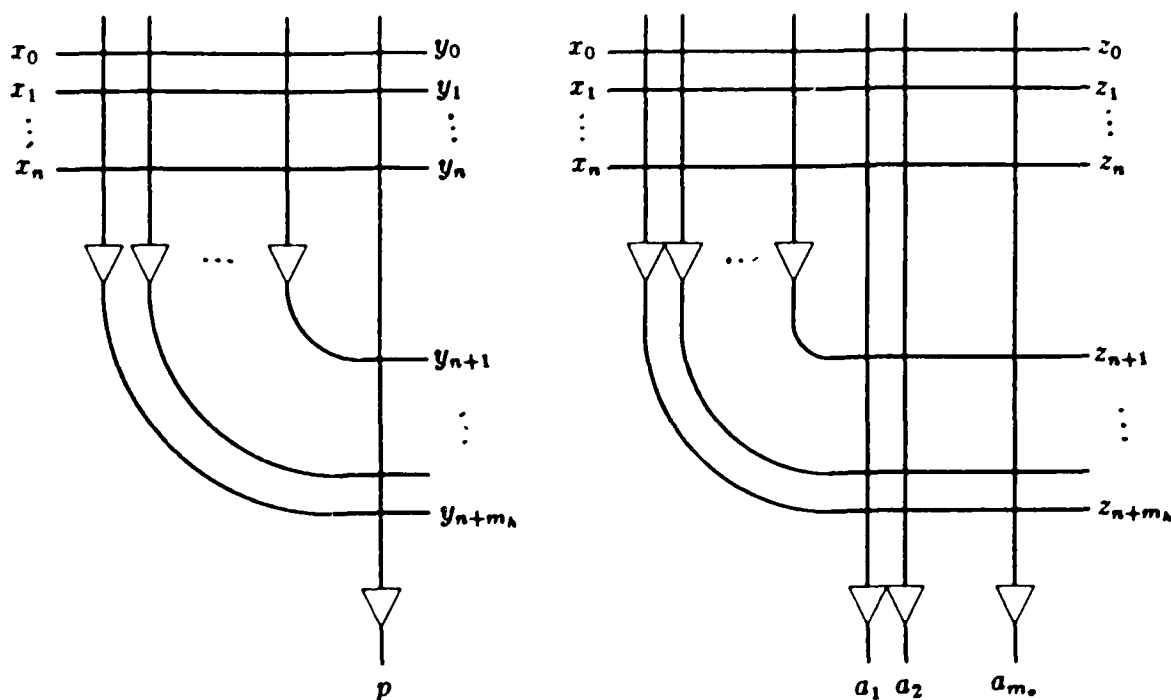


Figure 16: Two-Layer Networks for Strategy Learning

lines emanating from the apex of the triangles. Input from the environment at time step t , denoted by $x_0[t], x_1[t], \dots, x_n[t]$, is provided to all hidden units and output units. There is an interconnection weight at every intersection—hidden units receive $n + 1$ inputs and have $n + 1$ weights each, whereas output units receive $n + 1 + m_h$ inputs and have $n + 1 + m_h$ weights. For the evaluation network, the weight associated with the i^{th} input to unit j at time step t is denoted $v_{ij}[t]$; the analogous weight of the action network is denoted $w_{ij}[t]$.

The learning rule for the evaluation network is composed of Sutton's [47] Adaptive Heuristic Critic (AHC) method for the output unit and Rumelhart, Hinton, and Williams's [44] error back-propagation scheme for the hidden units. The AHC rule results in a prediction of future reinforcement for a given state. Changes in this prediction are used as *heuristic reinforcement* to guide the learning of search heuristics by the action network. The output units of the action network use an associative reinforcement-learning method identical to the one used in our earlier pole-balancing paper [13, 47, 45]. Other associative reinforcement-learning methods, such as the Actor-Critic rule, could be used for the output unit. The Actor-Critic rule would require

an additional mechanism for restricting the heuristic reinforcement to between 0 and 1. We chose to draw on our previous experience with the single-layer network [13] by employing the reinforcement-learning rule used there. This rule is combined with the error back-propagation method for hidden units.

Output Functions

Evaluation Network The output of the evaluation network is computed in the following way. First, the outputs of the hidden units are calculated. The output, p_j , of hidden unit j is calculated using the values of its weights, v , at time t_v and input, x , at time t_x , as follows:

$$p_j[t_x, t_v] = f\left(\sum_{i=0}^n x_i[t_x] v_{i,j}[t_v]\right), \quad \text{for } j \in H,$$

where f is the logistic function $f(s) = 1/(1 + e^{-s})$. The multiplication of weight and input vectors from different time steps is required by the learning rules for reasons described below.

The input vector, y , for the output unit is composed of the input from the environment and the output of the hidden units:

$$y_i[t_x, \cdot] = x_i[t_x], \quad \text{for } i = 0, \dots, n,$$

$$y_i[t_x, t_v] = p_{i-n}[t_x, t_v], \quad \text{for } i = n+1, \dots, n+m_h.$$

The index, m , of the single output unit is dropped from p_m for clarity. Thus, the output of the evaluation network is p , and is defined as

$$p[t_x, t_v] = \sum_{i=0}^{n+m_h} y_i[t_x, t_v] v_{i,m}[t_v].$$

Action Network To define the output of the action network we first define the hidden unit outputs, a_j :

$$a_j[t] = f\left(\sum_{i=0}^n x_i[t] w_{i,j}[t]\right), \quad \text{for } j \in H.$$

These values partly determine the input vector, z , for the output units, along with the input from the environment:

$$\begin{aligned} z_i[t] &= x_i[t], & \text{for } i = 0, \dots, n, \\ z_i[t] &= a_{i-n}[t], & \text{for } i = n+1, \dots, n+m_h. \end{aligned}$$

For the experiments in later chapters, the output components, a_j , $j \in O$, of the action network are in one-to-one correspondence with the possible actions defined for the task.¹ To select an action for a given problem state, the output of one output unit is set to 1 and the outputs of other units are set to 0 by the following process. The output functions of the reinforcement-learning units are stochastic, i.e., their output depends on a noisy weighted sum of inputs. A competition among the output units is implemented by assigning the value 1 to the unit with the highest weighted sum plus noise. This competition is limited to units corresponding to legal actions for the current state. Let $L_t \subseteq O$ be the set of indices for the output units that represent legal actions for the state at time t . The determination of L_t at each time step can be implemented by a network and even learned through experience, though for our experiments we specified L_t a priori. The responses of the output units are calculated as follows.

Let s_j be the noisy weighted sum of the input for unit j , $j \in L_t$, defined as

$$s_j[t] = \sum_{i=0}^{n+m_h} z_i[t] w_{i,j}[t] + \eta_j[t],$$

where $\eta_j[t]$ is random variable with distribution function Ψ (for the pole-balancing task, Ψ is the logistic distribution). The unit with the largest value for s_j wins the

¹ Rather than representing actions in this localized way, each action can be encoded by a pattern of output-unit activity. For example, the six possible actions for the Tower of Hanoi puzzle could be represented as patterns of output values over three output units. This can lead to generalization among actions represented by similar output patterns, which can either benefit or hinder the learning of correct actions. These issues are not addressed by the representation described in the text, although the reinforcement-learning method is capable of dealing with the credit-assignment problem that results when output patterns encode actions.

competition and is assigned a nonzero output:

$$a_j[t] = \begin{cases} 1, & \text{if } s_j[t] > s_k[t], \text{ for } k \in L_t \text{ and } j \neq k; \\ 0, & \text{otherwise.} \end{cases}$$

To simplify the determination of the unit with the largest s_j for the Tower of Hanoi task, the following exponential probability distribution is used for Ψ :

$$\Psi(q) = 1 - e^{-q}.$$

The output function can be simplified for tasks with only two possible actions for every state, such as the pole-balancing task. A single output unit is used whose binary output values encode the two actions. Let this unit be unit k , i.e., $O = \{k\}$. The specialization of the output function for this case is:

$$a_k[t] = \begin{cases} 1, & \text{if } s_k[t] > 0; \\ 0, & \text{otherwise.} \end{cases}$$

Learning Rules

Output Layer of Evaluation Network—The change in p plus the value of the external reinforcement r is called the heuristic reinforcement, or \hat{r} :

$$\hat{r}[t] = r[t] + \gamma p[t, t-1] - p[t-1, t-1],$$

where $0 < \gamma < 1$, called the *discount rate*. The use of \hat{r} in updating the weights of the evaluation network's output unit results in a prediction of future discounted reinforcement for the current state, with reinforcement farther in the future discounted more than earlier reinforcement [47,46]. States for which p is relatively large are favorable, while those with relatively low p are to be avoided. Once this mapping is correctly formed, changes in p can be used to indicate whether recent actions are leading toward favorable or unfavorable states.

The double time dependencies of variables in the equations for the evaluation network are needed for the following reason. In comparing one value of p with a

previous value, care must be taken to avoid instability in the growth of weight values (equations for changing weight values are presented shortly). If the computation of p for step $t - 1$ uses $v[t - 1]$ whereas p for step t uses $v[t]$, then a change in p from one time step to the next could be caused by a change in weight values rather than the encounter of a state with a different expectation of reinforcement. To avoid this, the pair of subsequent p 's is based on a single set of weight values, i.e., the difference between p for step $t - 1$ and for step t is due only to the change from $x_i[t - 1]$ to $x_i[t]$, because both p 's are calculated using $v_i[t - 1]$. If weights are known to change by small magnitudes on each step, then this precaution may not be necessary (as done in Ref. [13]).

Sutton [47] specialized the AHC rule by redefining \hat{r} for several classes of tasks involving distinct *trials*, where a trial consists of the following steps:

1. setting the state of the problem to a *start state*,
2. letting the learning system and environment interact, until
3. a *goal state* or *failure state* is encountered, signaled by a particular external reinforcement value.

Following Sutton, \hat{r} for trial-based tasks, such as those considered here, is defined to be:

$$\hat{r}[t] = \begin{cases} 0, & \text{if state at time } t \text{ is a} \\ & \text{start state;} \\ r[t] - p[t - 1, t - 1], & \text{if state at time } t \text{ is a goal} \\ & \text{or failure state;} \\ r[t] + \gamma p[t, t - 1] - p[t - 1, t - 1], & \text{otherwise.} \end{cases} \quad (5.1)$$

The weights of the output unit, unit m , of the evaluation network are updated by the following equation:

$$v_{i,m}[t] = v_{i,m}[t - 1] + \beta \hat{r}[t] y_i[t - 1, t - 1],$$

for $i = 0, \dots, n + m_h$ and $\beta > 0$. A positive change in state evaluations, indicated by a positive \hat{r} , results in an increase (decrease) in weight values proportional to

the corresponding positive (negative) input values on the preceding steps. In this way, the evaluation of the preceding state is altered, effectively shifting evaluations to earlier states.

The above expression is a simplification of Sutton's learning rule: in its general form, $y_i[t-1, t-1]$ is a trace of previous values of y_i , called an *eligibility trace*. An example of an eligibility trace is a weighted average of past values of y_i with recent values weighted more heavily. This generally results in faster development of good evaluation functions. Eligibility traces can also be used in the weight update equations of the action network. We chose not to implement eligibility traces primarily for the following reason. Preliminary experiments with the pole-balancing task showed that a one-layer action network functioning with eligibility traces and without an adaptive evaluation network, i.e., learning only from the external reinforcement, could learn to perform relatively well. However, our interests were in studying learning in hidden units, which are required for the development of a good evaluation function for the pole-balancing task as it is formulated here. We removed the eligibility traces from both networks to force a greater reliance on the evaluation function and to increase the number of failures early in a run, providing more external reinforcement and thus more opportunities to improve the evaluation function. Thus, our primary goal was not to achieve the fastest possible learning on this task but to investigate learning in hidden units.

Output Layer of Action Network Output unit j , $j = L_A$, of the action network updates its weights according to:

$$w_{i,j}[t] = w_{i,j}[t-1] + \rho \hat{r}[t] (a_j[t-1] - E\{a_j[t-1]|w; z\}) z_i[t-1],$$

for $i = 0, \dots, n + m_h$, where $E\{a_j[t-1]|w; z\}$ is the expected value of $a_j[t-1]$ conditional on the current values of w and z . Weight values are not changed for output units corresponding to illegal actions. The value of $a_j[t-1] - E\{a_j[t-1]|w; z\}$ can be viewed as a measure of the difference between action $a_j[t-1]$ and the action that is usually taken for the given values of $z_i[t-1]$ and $w_{i,j}[t-1]$. Thus, the results of an unusual action have more of an impact on the adjustment of weights than do other actions. Since $a_j \in \{0, 1\}$, the expected value of a_j is equal to the probability that

a_j is 1, i.e.,

$$E\{a_j[t]|w; z\} = Pr\{a_j[t] = 1\}.$$

See Ref. [4] for derivations of this probability for the case of three actions — the only cases that arise for the formulation of the Tower of Hanoi puzzle used in Section 6. The calculation of this probability is easy for the pole-balancing task because there are just two possible actions. In this case, $Pr\{a[t] = 1\}$ is just $\Psi(q)$, where q is the weighted sum of the unit's input.

Hidden Layer of Evaluation Network From the results of the comparative experiments described in Section 4, we concluded that the error back-propagation method of Rumelhart, Hinton, and Williams usually learned most rapidly (for the particular multiplexer task used in the experiments). However, this method cannot be applied directly because it requires knowledge of the correct output. Here we do not know the correct action or the correct evaluation for a given state, which would be needed in order to calculate an error to be back-propagated.

To apply an error back-propagation scheme to the hidden units of a network whose output layer is learning through reinforcements, a way of translating a reinforcement into an error must be found. This can be done in a heuristic manner by extracting from the reinforcement-learning equations the terms that govern weight updates in a fashion similar to the error terms in the gradient-descent rules. However, it is not obvious how to incorporate the eligibility traces often used in reinforcement-learning methods into a back-propagation scheme (this is another reason for not including traces for the experiments reported here).

For the evaluation network, \hat{r} plays the role of an error in the update of the output unit's weights. Therefore, we define the error of the output unit, δ_m^p , to be:

$$\delta_m^p(t) = 1 - \hat{r}(t),$$

where the superscript denotes the association with the evaluation network that generates output p . The error that is back-propagated from the output unit to hidden unit j is just \hat{r} , and Rumelhart et al.'s [44] expression with Sutton's [48] modification for the error of hidden unit j , called δ_j^p , becomes:

$$\delta_j^p(t+1) = \delta_m^p(t+1) \text{sgn}(v_{j+n,m}(t+1)) y_j(t+1, t+1) (1 - y_j(t+1, t+1)),$$

and their method for updating the hidden units' weights can be applied:

$$v_{i,j}[t] = v_{i,j}[t-1] + \beta_h \delta_j^p[t-1] x_i[t-1] + \beta_m \Delta v_{i,j}[t-1],$$

for units $j \in H$ and inputs $i = 0, \dots, n$. Note that the sign of hidden unit j 's output weight rather than the weight value itself is used. This variation is used because the results of the comparative study reported in Section 4 suggest that the method's sensitivity to the value of the learning rate parameter, here β_h , is decreased by the use of the sign of the weight.

Hidden Layer of Action Network—The equation for updating the weights of the action network's hidden units is a bit more complicated. Once \hat{r} becomes a good evaluation of the previous action, the role of an error is played by the product of \hat{r} and the difference between the previous action and its expected value. The sign of the product is an indication of whether the action probability should be increased or decreased. So the error in the output of output unit k , $k \in L_t$, of the action network is defined as:

$$\delta_k^a[t-1] = \hat{r}[t] (a_k[t-1] - E\{a_k[t-1]|w;z\}).$$

The back-propagated error to hidden unit j is used to compute the hidden unit's error:

$$\delta_j^a[t-1] = \sum_{k \in L_t} (\delta_k^a[t-1] \text{sgn}(w_{j+n,k}[t-1])) z_j[t-1] (1 - z_j[t-1]),$$

and the weights are updated by the following equation:

$$w_{i,j}[t] = w_{i,j}[t-1] + \rho_h \delta_j^a[t-1] x_i[t-1] + \rho_m \Delta w_{i,j}[t-1],$$

for units $j \in H$ and inputs $i = 0, \dots, n$. Disregarding the different errors that are back-propagated by the two networks, the learning rule used by the hidden units of the two networks are identical. The sum over the products of output unit errors and weights is not included in the expression for a hidden unit's error in the evaluation network because there is only one output unit.

Parameters The equations for the evaluation network are governed by the follow-

ing parameters:

$$\begin{aligned}\beta &= \text{learning rate for the output unit} & (\beta > 0); \\ \beta_h &= \text{learning rate for the hidden units} & (\beta_h > 0); \\ \beta_m &= \text{momentum factor for the hidden units} & (\beta_m \geq 0); \\ \gamma &= \text{discount rate} & (0 \leq \gamma < 1).\end{aligned}$$

Similar parameters appear in the equations for the action network:

$$\begin{aligned}\rho &= \text{learning rate for the output units} & (\rho > 0); \\ \rho_h &= \text{learning rate for the hidden units} & (\rho_h > 0); \\ \rho_m &= \text{momentum factor for the hidden units} & (\rho_m \geq 0).\end{aligned}$$

In applying this system to a task, it is important to test a number of values for each parameter to investigate the sensitivity of the methods with respect to the parameters. This was done for all experiments described in this report.

The Pole-Balancing Task

In this section we describe the pole-balancing task, our computer simulation of it, and how this simulated system interacts with the adaptive networks. Additional details can be found in Ref. [13]. Learning to solve the version of the pole-balancing, or inverted-pendulum, task that we have studied is nontrivial for two reasons:

1. the evaluation function to be learned is nonlinear and therefore cannot be formed by a single linear unit, and
2. a performance evaluation in the form of a failure signal appears only after a sequence of actions has been taken, making it difficult to identify which actions are good and which are bad.

The pole-balancing task involves a pole hinged to the top of a wheeled cart that travels along a track (as described in Ref. [13]). Both pole and cart are constrained to move in a plane. The state at time t of this dynamical system is specified by four

real-valued variables:

- x_t = the horizontal position of the cart, relative to the track;
- \dot{x}_t = the horizontal velocity of the cart;
- θ_t = the angle between the pole and vertical, clockwise being positive;
- $\dot{\theta}_t$ = the angular velocity of the pole.

The goal is to exert a sequence of forces, F_t , upon the cart's center of mass such that the pole is balanced for as long as possible and the cart does not hit the end of the track. More abstractly, the state of the cart-pole system must be kept out of certain regions of the state space, making this an avoidance control problem. There is no unique solution—any trajectory through the state space that does not pass through the regions to be avoided is acceptable. A minimal amount of knowledge about the task is assumed in our experiments. The only information regarding the goal of the task is provided by the external reinforcement signal, r_t , that signals the occurrence of a failure caused either by the pole falling past a prespecified angle, or the cart hitting the bounds of the track. r_t is defined as

$$r_t = \begin{cases} 0, & \text{if } -0.21 \text{ radians} < \theta_t < 0.21 \text{ radians and } -2.4 \text{ m} < x_t < 2.4 \text{ m;} \\ -1, & \text{otherwise.} \end{cases}$$

Note that r_t does not depend on $\dot{\theta}_t$ or \dot{x}_t .

We simulated the cart-pole system using a numerical approximation of the system of nonlinear differential equations that models the system. Details are provided in Ref. [13].

From successful experiments with two-layer systems, we found that good evaluation and action functions look like those sketched in Fig. 17.² For clarity, let us limit attention to projections of the functions to the $(\theta, \dot{\theta})$ subspace. Figure 17a shows the kind of function one might expect the evaluation network to learn. Failure is likely to occur in the upper right and lower left portions of the $(\theta, \dot{\theta})$ state space. We want

²Due to the nature of our formulation of the learning task as an avoidance control problem, one cannot say that these functions, or any others, are the unique optimal functions. There are very many ways that the system can avoid the failure regions of state space.

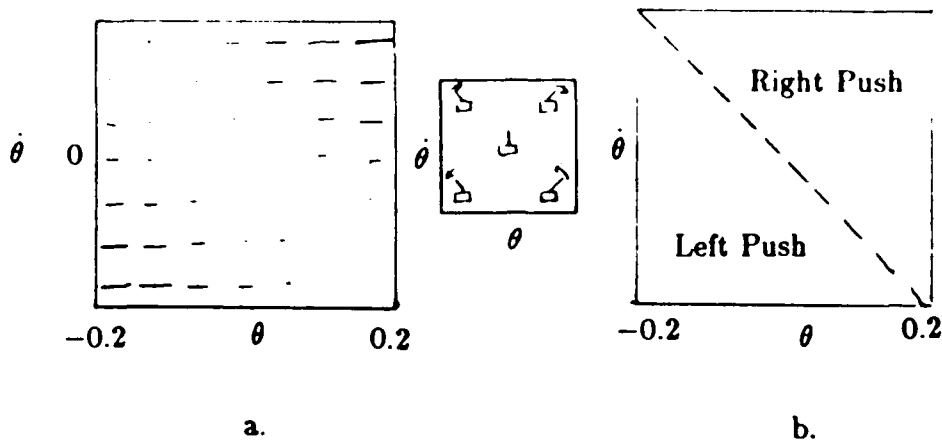


Figure 17: Good Functions for Pole-Balancing Solution

the learning system to shift this failure signal to states that precede failure states, then to states that precede failure by longer time intervals, with the strength of the shifted prediction indicative of the average number of time steps until failure. Past states and actions, or weighted averages of previous states and actions, could be used to apportion blame for the failure to previous actions, but the tradeoff between a) the need for a long history to blame actions many steps in the past, and b) the need for a short history to avoid blame being spread too thinly (resulting in slow learning), is difficult to optimize. These temporal credit-assignment issues are studied in detail by Sutton [47,46], who developed the AHC learning rule used here in the output unit of the evaluation network.

The map from (x, \dot{x}) to a prediction of failure also looks like Fig. 17a. In the lower left corner, the cart is moving to the left and is near the left border of the track, and in the upper right corner it is approaching the right border of the track.

Figure 17b shows an action function for generating a push on the cart. For small angles, such as $-0.21 < \theta < 0.21$ as used in our experiments, the surface that separates states requiring different actions, called the *switching surface*, can be linear. States in the upper right region require a push to the right, while states in the lower left require a left push. The linear switching surface is an approximation to the nonlinear switching surface of the time optimal bang-bang controller. The linear approximation works well for the small range of angles used in the experiments. The position and slope of the linear surface varies for different values of x and \dot{x} .

The pole-balancing task frequently has been used to illustrate standard control techniques due to the inherent instability of the pole and the task's similarity to many balance-control problems. For example, Cannon [17] shows how the root-locus method is used to analyze the stability of a "lead compensation-network" controller that exerts a force proportional to the derivative of an error signal in this case the pole's angular velocity. His analysis is confined to small angles and angular velocities and does not include the goal of avoiding the bounds of the track. These control-design techniques require a model of the system to be controlled in the form of differential equations that define how the state variables change over time. A good deal of time must be spent by the control engineer in determining a model that approximates the behavior of the system to the desired degree of accuracy. Control systems that learn without a predefined model, or that acquire internal models through observation of the system's behavior, would obviate this potentially difficult analysis.

An alternative to expressing a control law as an analytical equation is to represent the function in tabular form. Michie and Chambers [35] took this approach for their learning system as applied to the pole-balancing task. Their table consisted of approximately 162 "boxes" – nonoverlapping, rectangular regions of the cart-pole system's state space – containing average counts of the number of steps before failure for a push to the right when the system's state addresses the corresponding box, and an analogous count for a push to the left. When a box is entered, the push with the highest count is applied. Their system successfully improved its performance with experience.

Our previous learning system for this task [13] integrated the table look-up approach with connectionist learning methods. Separate tables were used to store predictions of reinforcement and probabilities of generating actions, each indexed by the state of the cart-pole system. The tables were implemented as two units, each receiving 162 binary-valued input components. When the state was in a particular box, the corresponding input component was set to 1 and all other components were set to 0. Therefore, the weighted sum of the unit's input was equal to the value of the weight associated with the nonzero input component.

An obvious problem with the table look-up approach is that the size, shape, and placement of the regions into which the state space is divided greatly influence the ability of the system to learn the desired mappings. A region might be too large, meaning that different states inside the region require different output values, e.g., different pushes on the cart. Conversely, regions are smaller than optimal when many regions require the same output. If these regions are instead subsumed by one large region, then what is learned for one state is generalized correctly to all other states in the region. With many small regions, learning must occur in all regions independently. Michie and Chambers proposed a solution to this problem: regions for which one output is not clearly better than any other despite repeated experience should be "split" into several, smaller regions, and regions with the same output value should be "lumped" into a single, larger region. Politis and Licata [39] have pursued this possibility with Barto, et al.'s [13] learning system and a technique for periodically splitting every region uniformly into a number of smaller regions.

In the experiments described here, the fixed "decoder" of the system described in Ref. [13] to translate the cart-pole state to a region address is replaced by a layer of hidden units that learns features useful in solving the pole-balancing task. This "adaptive decoder" view is closely related to current research topics in control theory involving the application of multiple controllers to one task. For example, the control of a full 360-degree pole requires a complex control law in order to be useful for all states. An alternative is to use a collection of less-complex control laws, and activate one at a time, based on the current state and an ordering of the control laws according to their ability to deal with that state. Learning when to switch from one controller to another is analogous to learning how to classify the state into one of a set of boxes.

Another example of a connectionist-style system applied to the pole-balancing task comes from the work of Widrow and Smith [60]. They present results of using a supervised-learning scheme to train a network of Adaline units to duplicate the responses of a teacher. For their experiments the teacher was a predefined linear control law. A human could play the role of the teacher by manually controlling the pole through an interface, such as a joystick, and the Adaline network could use the

human's responses as training examples. Learning to mimic a teacher is much easier than learning from delayed reinforcement as is required for our formulation of the pole-balancing task, but the work of Widrow and Smith has been very useful to us in showing how adaptive units can be applied to control problems.

Interaction between the Network and Cart-Pole Simulation

The components of the networks' input, $x_i[t]$, are scaled versions of the state variables:

$$x_1[t] = \frac{1}{4.8}(x[t] + 2.4),$$

$$x_2[t] = \frac{1}{3}(\dot{x}[t] + 1.5),$$

$$x_3[t] = \frac{1}{0.42}(\theta[t] + 0.21),$$

$$x_4[t] = \frac{1}{4}(\dot{\theta}[t] + 2).$$

An additional input, $x_0[t]$, with a constant value of 0.5 provides a variable threshold. Inputs $x_1[t]$ and $x_3[t]$ range from 0 to 1, while $x_2[t]$ and $x_4[t]$ are primarily within the 0-1 range, but can fall outside these bounds. This scaling accomplishes two things. Since the learning rules involve the input terms, $x_i[t]$, as factors in the equations for updating weight values, terms with predominantly larger magnitudes will have a greater influence on learning than will other terms. To remove this bias all input terms are scaled to lie within the same range. Secondly, since the values of the state variables are centered at zero, if these values were used directly as network inputs the correct action for positive θ and $\dot{\theta}$ would transfer to negative θ and $\dot{\theta}$ in exactly the right way, i.e., the correct action for negative θ and $\dot{\theta}$ is the negative of the action for positive θ and $\dot{\theta}$ (see Figure 17b). These correct generalizations make the task much easier, circumventing the need for a complex action network. Thus, scaling the state variables allowed a simple action network of hidden units in the case of the action network.

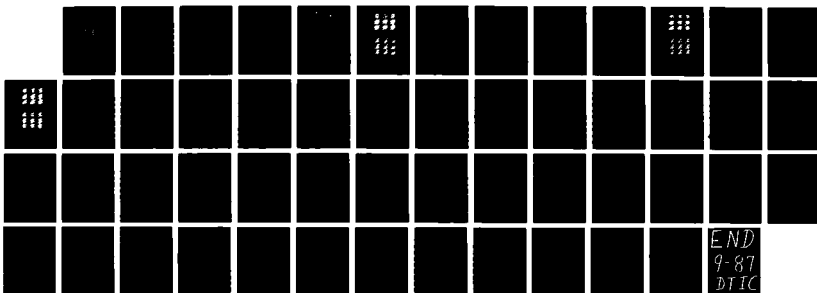
AD-A183 782

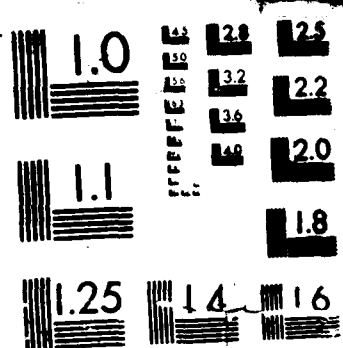
MULTILAYER NETWORKS OF SELF-INTERESTED ADAPTIVE UNITS
(U) MASSACHUSETTS UNIV AMHERST DEPT OF COMPUTER AND
INFORMATION SCIENCE A G BARTO JUL 87 AFMIL-TR-87-1852
F33615-83-C-1078 F/G 12/77

2/2

UNCLASSIFIED

NL





MICROCOPY RESOLUTION TEST CHART

The force exerted on the cart's center of mass at time t is given by:

$$F_t = \begin{cases} 10 \text{ nt,} & \text{if } a[t] = 1; \\ -10 \text{ nt,} & \text{if } a[t] = 0, \end{cases}$$

where $a[t]$ is the binary-valued output of the action network at time t . The sampling rate of the cart-pole system's state and the rate at which control forces are applied are the same as the basic simulation rate, i.e., 50 hz..

Results

The experiments consisted of a number of *trials*, each starting with the cart-pole system set to a state chosen at random, and ending with the appearance of the failure signal. A series of trials constitutes a *run*, with the first trial of a run starting with weights initialized to random values between -0.1 and 0.1 . We want the learning system to learn to generate actions, $F[t]$, that maximize the number of time steps between occurrences of $r[t] = -1$. The only information available to the system is given by the sequences $x_i[t]$, $i = 0, \dots, 4$, and $r[t]$.

One-Layer Experiments

We experimented with one-layer networks (no hidden units) to obtain performance measures with which the performance of the two-layer system could be compared. The learning rules for the one-layer networks depend on the three parameters, ρ , β , and γ . The value of γ was fixed at 0.9 , while different values of ρ and β were crudely optimized (simulation time prevented an accurate optimization) by performing 2 runs of 500,000 steps each for approximately 25 different sets of parameter values. Two performance measures were used to select the best parameters. The number of trials, averaged over runs with one set of parameter values, provides a rough measure of performance over the length of a run. To judge how well the solution had been learned by the end of the run, the number of steps in the last trial, or the previous trial, whichever is larger, is averaged over all runs. In this way, an

abnormally short final trial caused by the termination of a run on the 500,000th step does not enter into the average of final trial lengths.

Performance did not vary considerably for the parameter values that were tested. The best parameter values were used to obtain a more statistically-significant result by performing 10 runs of 500,000 steps each. The following parameter values were used:

$$\begin{aligned}\beta &= 0.05, \\ \rho &= 0.5, \\ \gamma &= 0.9,\end{aligned}$$

resulting in the number of failures for the 10 runs shown in Table 14. The average

Table 14: Results of One-Layer System

Run	Trials	Last Trial
1	33,977	14
2	61,888	4
3	24,795	16
4	22,717	130
5	28,324	28
6	15,218	100
7	31,594	15
8	44,903	9
9	16,115	72
10	26,402	14

number of trials for each run is approximately 30,593. In addition, the number of steps in the last trial is shown for each run. As explained above, this value is actually the larger of the last trial length and the previous trial length, in case the last trial had just begun when the run was terminated at step 500,000.

The number of steps per trial versus the number of trials is plotted in Fig. 18. The plotted values are averages over the 10 runs and over bins of 100 trials, i.e., the trials for a run are grouped into intervals of 100 trials, the number of steps per trial is averaged for each interval, and the results are averaged over the runs. The learning curve shows that performance improves with experience—the trial length is approximately equal to 10 steps initially, and after 30,000 trials approximately 30

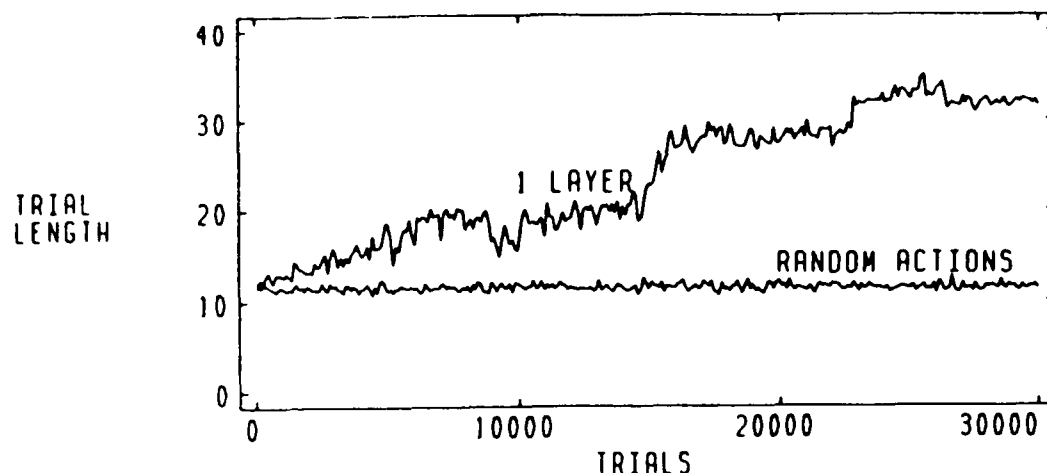


Figure 18: Balancing Time versus Trials for One-Layer System

steps occur per trial. Even with little experience, the learning system performs better than a fixed controller that selects pushes on the cart at random. The large variance from trial to trial is due to the initialization of the cart-pole system to random states upon failure. The starting state of each trial might be very similar to a failure state, or it might be near the state of perfect balance where $(x, \dot{x}, \theta, \dot{\theta}) = (0, 0, 0, 0)$. This method of restarting after failure differs from that used in our earlier work described in Ref. [13], where we started the cart-pole system at state $(0, 0, 0, 0)$ after every failure.

The values of the weights at the end of each run varied considerably. The best of the 10 runs, resulting in 15,218 failures, resulted in the weights that are displayed on the network schematic shown in Fig. 19. Positive weights are drawn as hollow circles and negative weights as filled disks. The magnitude of a weight is proportional to the radius of its circle, or disk. From the size of the weights we see that the output of the evaluation network (Fig. 19a) is rather insensitive to the values of the state variables, and the value of the output is always negative. The output of the action network (Fig. 19,b) does depend on the system's state. A large θ has a positive effect, producing a push on the cart to the right, and a large value for x has a negative effect, pushing the cart to the left.

A better understanding of what these weights mean is obtained from a graph of the output of the networks versus the state. To display these functions of four

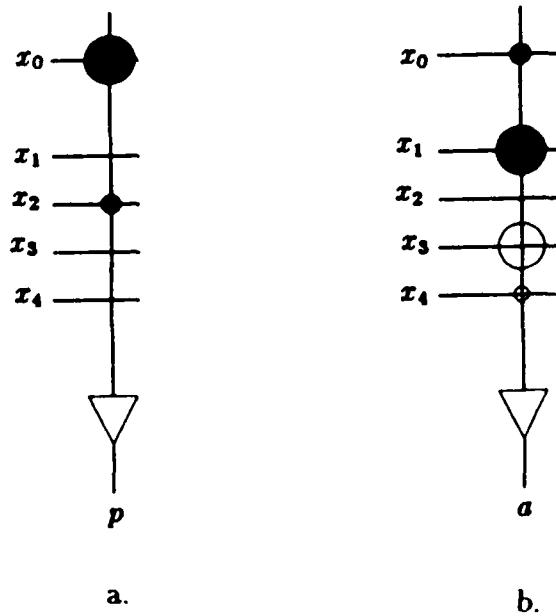
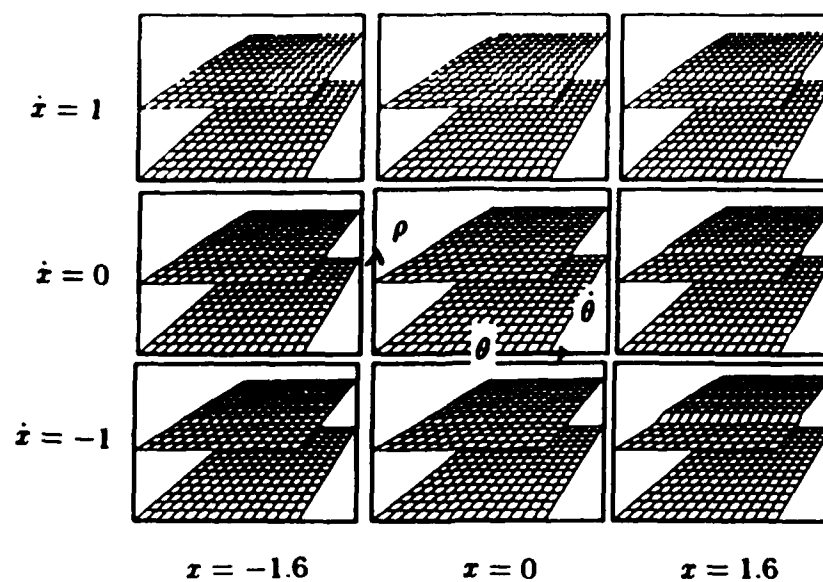


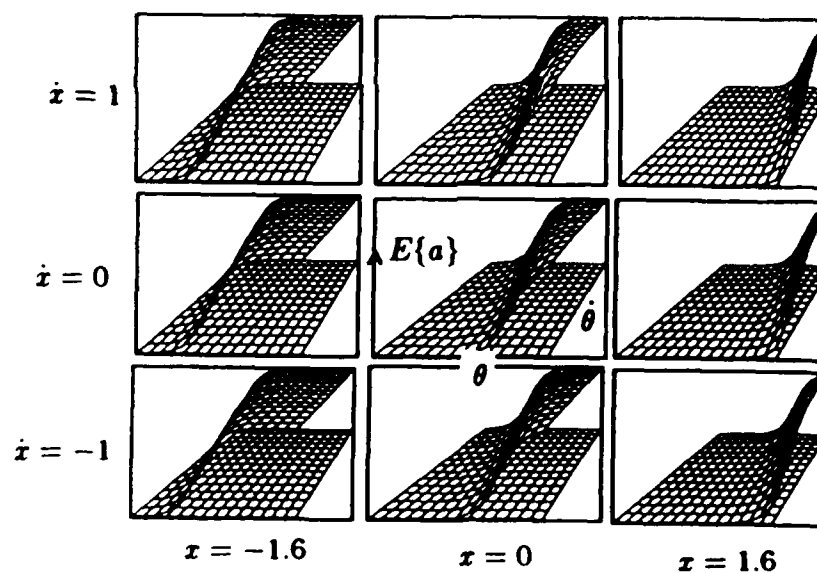
Figure 19: Weights Learned by One-Layer Network

variables, we generated graphs of the functions' values versus θ and $\dot{\theta}$ for nine different pairs of x and \dot{x} values. Fig. 20a and Fig. 20b contain such graphs for the evaluation network and the action network, respectively. The insensitivity of the evaluation network to the state is evident from the flat surfaces of its graphs. The base plane in these graphs does not represent a value of 0; the surface is actually at a small negative value. Obviously this function serves no useful role as an evaluation function for states. It is for this reason that the one-layer system could not improve its performance over 30 steps per trial. Credit is assigned by the external reinforcement signal only to actions that push the cart-pole into a failure state in one step. These actions may not be responsible for the failure and may even be correct.

Fig. 20b shows that the action network has learned a function with approximately the desired shape (Fig. 17b). The height above the base plane represents the probability of generating a push to the right. The level of the base plane is at zero probability, so for states where the surface lies near the base plane, a push to the left is generated with high probability. The middle graph, where $x = 0$ and $\dot{x} = 0$, shows a smooth transition from a high probability of pushing left to a high probability of pushing right as θ and $\dot{\theta}$ go from negative to positive. This transition is shifted in



a.



b.

Figure 20: Functions Learned by One-Layer Networks

the direction of negative θ for negative values of x and in the positive θ direction for positive values of x . In relating these graphs to those for the two-layer networks, we will see that this relationship between the transition line and the value of x is the opposite of what is needed to balance the pole while avoiding collisions with the track boundaries.

Results of Two-Layer Experiments

Two-layer networks were formed by adding 5 hidden units to each of the evaluation and action networks. The learning methods for the two-layer networks depend on seven parameters: β , β_h , β_m , ρ , ρ_h , ρ_m , and γ . As was done for the one-layer system, sets of parameter values (approximately 10) were each tested in 5 runs of 500,000 steps. Performance varied significantly for small changes in parameter values (γ was not varied). The values giving the best performance are:

$$\begin{aligned}\beta &= 0.2, \\ \beta_h &= 0.2, \\ \beta_m &= 0, \\ \rho &= 0.5, \\ \rho_h &= 0.5, \\ \rho_m &= 0, \\ \gamma &= 0.9.\end{aligned}$$

Notice that $\beta_m = \rho_m = 0$. Results suggest that nonzero momentum in the learning rules for the hidden units hinders performance on this task. These values were used for 10 runs of 500,000 steps, resulting in the total number of trials and final trial lengths shown in Table 15. The average number of trials over all runs is approximately 10,983, compared to 30,593 trials for the one-layer system. Even after much learning experience, a nonzero probability of selecting the wrong action exists for every state, as suggested by the relatively small number of steps in the last trials of Runs 1 and 10.

The learning curve for the two-layer system is shown in Fig. 21. The large, stair-like jumps in the curve are due to the way in which performance is averaged over runs, described as follows. The axis for the number of trials is labeled from 0 to 30,000 trials, so runs for which less than 30,000 trials occurred were handled in a special

Table 15: Results of Two-Layer System

Run	Trials	Last Trial
1	10,123	88
2	7,790	2,011
3	5,814	14,535
4	8,466	5,753
5	7,212	28,407
6	23,539	20,328
7	19,401	14,302
8	8,804	4,674
9	9,756	20,889
10	9,645	154

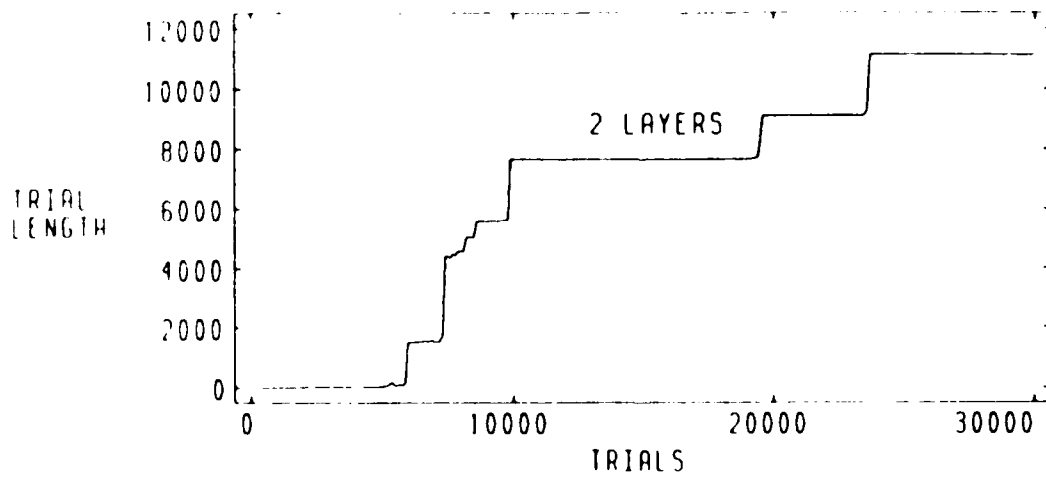


Figure 21: Balancing Time versus Trials for Two-Layer System

way. The learning curve for such a run is extended to 30,000 trials by assigning to trials that didn't occur a value equal to the larger of a) the number of steps in the last trial, and b) the number of steps in the previous trial. In this way, a very short final trial is disregarded and the length of the previous trial is used to extend the curve. The large jumps in the curve occur when the last trial of a particular run is very long and the run is terminated when 500,000 steps have elapsed. If the experiments were to be run for more steps, the final performance level would have been higher. This large number of steps is then averaged into the performance curve from that trial through trial 30,000. For example, the jump at trial 20,000 is due to the last trial of run 7, which was approximately 14,000 steps in length. All 10 runs were terminated before 30,000 trials elapsed, resulting in a final performance level of about 11,000 steps per trial. Recall that the final level achieved by the one-layer system was only about 30 steps per trial.

The large number of weights, 35 in each network, makes it difficult to interpret the solutions found directly from the weight values. The relative magnitudes and signs of the weights are shown in the network schematics of Fig. 22. Figure 22a shows the final weight values for the evaluation network of run 6, and Fig. 22b shows the weights for the action network. Units 1, 2, 4 and 5 of the evaluation network are similar, having all positive weights. (In the figure, the small size of the corresponding circles make them appear to be filled-in disks.) Unit 3's weights differ, and it is also distinguished by having a large positive connection to the output unit. It appears that only unit 3 has developed a new feature that is useful for the prediction of failure.

The function implemented by the evaluation network appears in Fig. 23a. The height of the surface ranges from approximately -1.5 and 0.1. Its shape is just what is needed for the action network to receive an immediate evaluation of an action. At the center of each base plane, representing the $(\theta, \dot{\theta})$ subspace, the cart-pole is in a state where the pole is vertical and not falling. The evaluation has its highest value for these states, therefore forming an evaluation function that decreases as the cart-pole system moves away from this state. Any action that takes the system toward either the positive or negative $(\theta, \dot{\theta})$ corner results in a negative evaluation change.

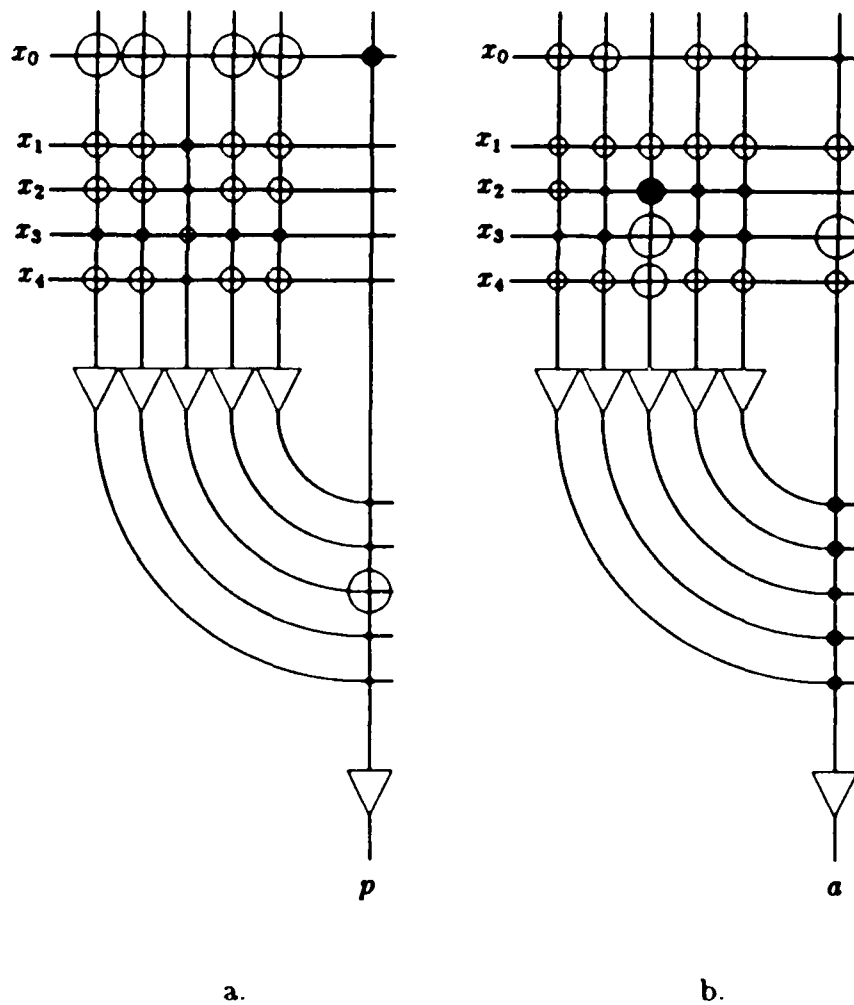
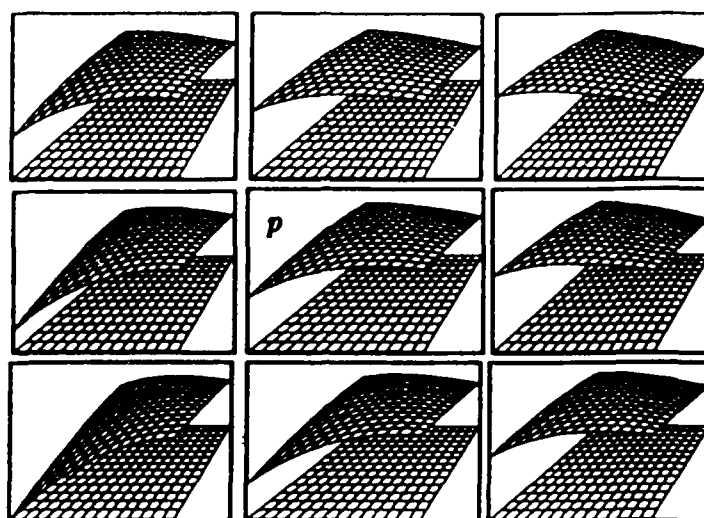
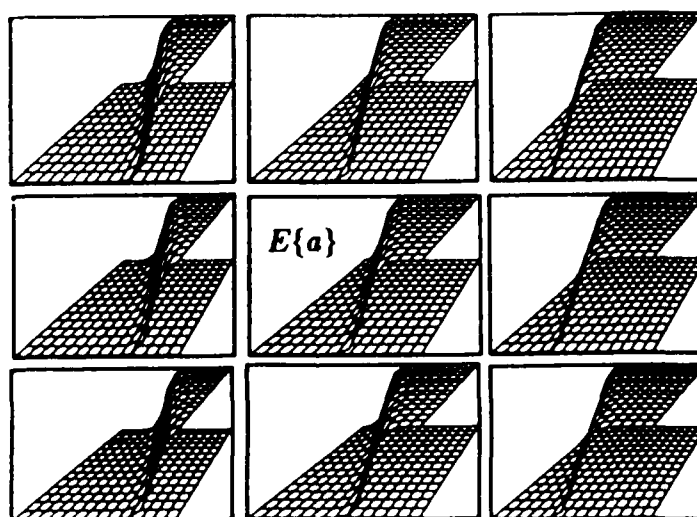


Figure 22: Weights Learned by Two-Layer Network



a.



b.

Figure 23: Functions Learned by Two-Layer Networks

i.e., a negative \hat{r} . The tilt of the surface as x and \dot{x} change is also correct. Positive $(\theta, \dot{\theta})$ states are more likely to result in failure when the cart is heading toward the right border of the track, where x and \dot{x} are positive. Similarly, negative $(\theta, \dot{\theta})$ states are likely to precede failure when the cart is heading to the left border, where x and \dot{x} are negative.

Before discussing the features learned by the hidden units, let us look at the solution learned by the action network. From Fig. 22b we see that again hidden unit 3 differs from the other units in its weight values: θ and $\dot{\theta}$ have large positive effects on unit 3's output, and x and \dot{x} have smaller positive and negative effects, respectively. Unit 3 is connected positively to the output unit, whereas the other units are connected negatively. The fact that the unit 3s of both networks play significant roles is fortuitous; for other runs useful features are learned by different sets of units.

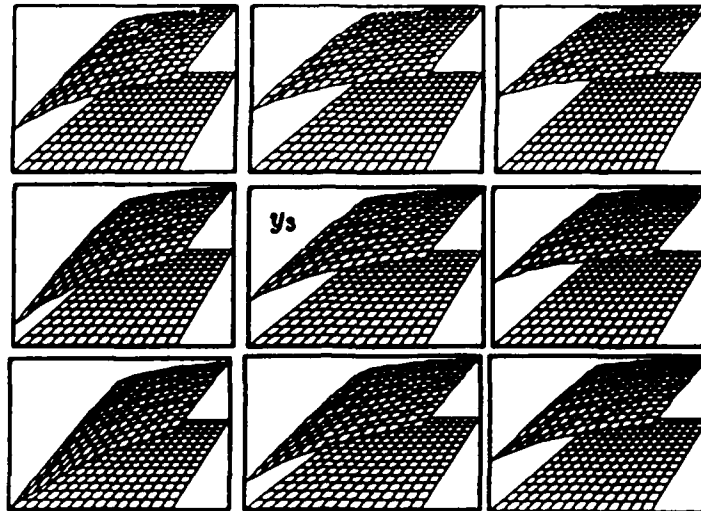
These hidden-unit influences in combination with the direct effects of the network's input on the output unit result in the action function displayed in Fig. 23b. Two observations can be made about the contrast between this action function and that learned by the one-layer network (Fig. 20b). First, the transition from a high probability of pushing left to a high probability of pushing right is much quicker, as θ and $\dot{\theta}$ vary. This probability function implements a much more deterministic control than does that of the one-layer network. Due to the good evaluation function learned by the evaluation network, actions near the transition line are credited or blamed appropriately. A second observation is that the shift in the transition line as x and \dot{x} vary is in the right direction. The pole should be balanced slightly to the right of vertical (positive θ) when the cart is near the left track boundary (negative x), and to the left of vertical when near the right boundary, resulting in a net action over several steps of a push towards the center of the track. To see that this is indeed what happens, note that the point at which the pole is balanced is roughly indicated by the location of the transition line. This line shifts toward positive θ when x is negative, and toward negative θ when x is positive.

Now we continue with the analysis of the hidden units. Unit 3 of both networks acquired significant effects on the respective output units. The functions implemented

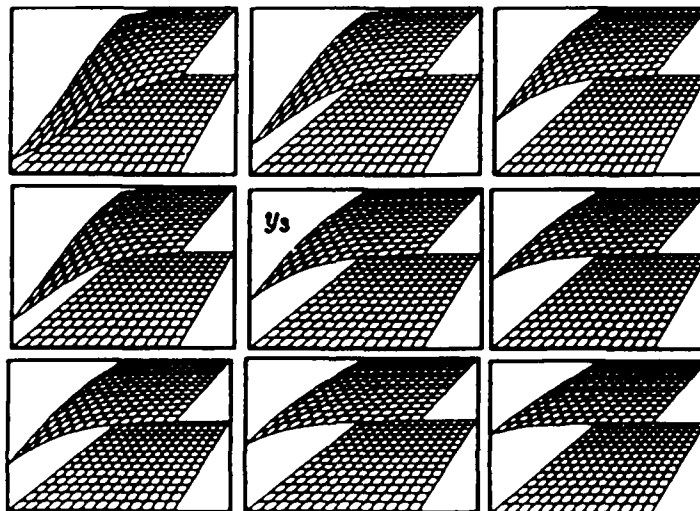
by their weights can be visualized by graphing them as functions of θ and $\dot{\theta}$ for different values of x and \dot{x} , as done for the functions implemented by an entire network. Figure 24a shows these graphs for unit 3 of the evaluation network and Fig. 24b shows the graphs for unit 3 of the action network. The outputs of these units varies between 0 and 1. Very similar functions were learned by the two units. They both produce a fairly constant value of 1 for most states, with lower values approaching 0 when θ and $\dot{\theta}$ become more negative. However, the contribution of unit 3 of the action network is very small—its output weight is small in comparison to the larger weights on the output unit, unit 6. This is not surprising, since the desired mapping from state to action can be implemented with a single unit. In fact, setting unit 3's output weight to 0 and adding its magnitude to unit 6's constant-input weight causes little change in the state-to-action mapping.

To test the significance of the new feature learned by unit 3 of the action network, further experiments were run with a one-layer action network and the two-layer evaluation network. The one-layer action network did learn the desired function, but it learned it more slowly than did the two-layer action network. Perhaps the feature learned by unit 3 facilitated the learning of a good action function, and with additional experience the output unit developed the appropriate weights for its state-variable inputs. This must be verified by observing the evolution of the action function both as a function of the state variables and as a function of the hidden units' outputs.

The role of unit 3 in the evaluation network is much more important. The hill-shaped evaluation surface cannot be implemented without the hidden units, as shown by the results of the single-layer experiments. Through its positively-weighted connection to the output unit, unit 3 generates the positive gradient in the evaluation surface as one moves from negative θ and $\dot{\theta}$ to $\theta = \dot{\theta} = 0$. At this point, the gradient in the response of unit 3 effectively becomes zero, and the output unit's negative weights from its state-variable inputs provide the negative slope as one moves in the positive $\theta, \dot{\theta}$ direction.



a.



b.

Figure 24: New Features Learned by Two-Layer Networks

Conclusion

It is immediately apparent from the learning curves of Figs. 18 and 21 that the two-layer learning system far outperformed the one-layer learning system. New features are required for representing a good evaluation function and, although they are not required for representing a good action function, new features facilitate learning the action function. Thus, the synthesis of the error back-propagation scheme of Rumelhart, Hinton, and Williams [44] and reinforcement-learning techniques produced an adaptive network that successfully deals with delayed reinforcement and the initial lack of an adequate representation. The learning method resulted in a controller that balanced the pole for 9 minutes (simulated time—28,000 steps at 0.02 seconds per step) and probably would have balanced it longer if the experiments had been run for a greater number of steps.

Comparison with the single-layer system of Barto et al. [13] is made difficult by the differences in how the experiments were conducted. The difference with the greatest effect is that in the experiments described in Ref. [13], the cart-pole system was always restarted in the same state, $(x, \dot{x}, \theta, \dot{\theta}) = (0, 0, 0, 0)$, following failure, whereas here the start state was selected randomly. Average performance is kept low by start states that are very close to failure states. Disregarding this difference, comparisons show that the previous system achieved a higher average trial length after 500,000 steps, 80,000 steps per trial, while the experiments here resulted in approximately 30,000 steps per trial. This difference reflects the fact that the two-layer system learned good solutions later in the runs than did the system of Ref. [13]. We conclude that a considerable number of steps are required for the hidden units to learn the necessary features. It is not until good features are learned that a useful evaluation function can be formed, and until the evaluation function is learned, the action network cannot improve beyond a low level of performance.

SECTION 6

LEARNING TO SOLVE A PUZZLE

Described in this section are experiments in applying a strategy learning network as described in Section 5 to the Tower of Hanoi puzzle. Because the state-space concept underlying the learning system for the pole-balancing problem also applies to problem-solving tasks that have been studied by Artificial Intelligence (AI) researchers, it is possible to use the same kind of learning methods for both types of problems. Applying this kind of learning method to a problem like the Tower of Hanoi puzzle is very different from applying one of the more knowledge-intensive learning methods of AI, and the results may not really be comparable. The knowledge-intensive approach is probably closer to how a human might learn to solve a puzzle like this, whereas the network method seems more similar to the acquisition of a motor-skill. The primary purpose in applying the network method to this puzzle was to provide a good test of the multilayer strategy learning network. Nevertheless, we still make some comparisons of methodology with Langley's [31] adaptive production system, called SAGE, which learns heuristics that improve the performance of an initial weak search strategy. To facilitate this comparison, we selected the three-disk Tower of Hanoi puzzle for our experiments, one of the puzzles that Langley used to demonstrate SAGE.

The Tower of Hanoi Puzzle

The Tower of Hanoi puzzle is popular for research in problem-solving because the number of states is small, but the puzzle is still difficult to solve. Human strategies for solving the Tower of Hanoi have been analyzed [32] and modeled [5]. Amarel [2]

used the Tower of Hanoi puzzle as a vehicle for studying shifts of representations to forms of increasing efficiency for the discovery of a problem's solution.

The state of the Tower of Hanoi puzzle can be represented in a number of ways. A common representation is one used by Nilsson [38], in which the pegs are numbered 1, 2, and 3, and the disks are labeled A, B, and C, where disk A is the smallest disk and C is the largest. A particular state is represented by the peg numbers where each disk resides, listed for disk C, then disk B and disk A. As pictured in Fig. 25, the initial state of the puzzle is (111), and the objective is to achieve state (333) by applying a sequence of actions. The legal actions are movements of the top-most

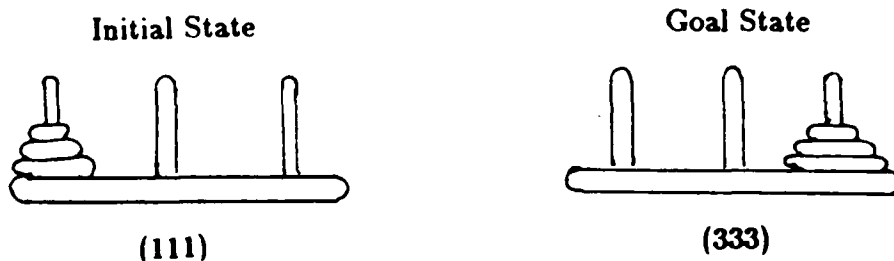


Figure 25: Initial and Goal States of the Tower of Hanoi Puzzle

disk from one peg to another, with the restriction that a disk may never be placed upon a smaller disk. An action may be represented as a source peg and destination peg, so the transformation of state (111) to state (112) is performed by the action of moving the top-most (smallest) disk from peg 1 to peg 2, represented by action 1-2. For the three-peg puzzle, six actions are needed: 1-2, 1-3, 2-1, 2-3, 3-1, and 3-2.

The states of the puzzle plus the transitions between the states corresponding to the legal actions form the puzzle's state transition graph shown in Fig. 26. To evaluate ability to improve search strategies on the Tower of Hanoi puzzle, we measure the number of actions in the solution path, with the minimum length path being the objective. For the three-disk puzzle, the minimum-length solution path has seven actions and is the straight path down the right side of the state transition graph. Finding the shortest solution path is confounded by the large number of possible solution paths and by the presence of loops in the state transition graph.

The Tower of Hanoi puzzle is a good test of the multilayer network described in Section 5 for the following reason. A useful evaluation function must map states to

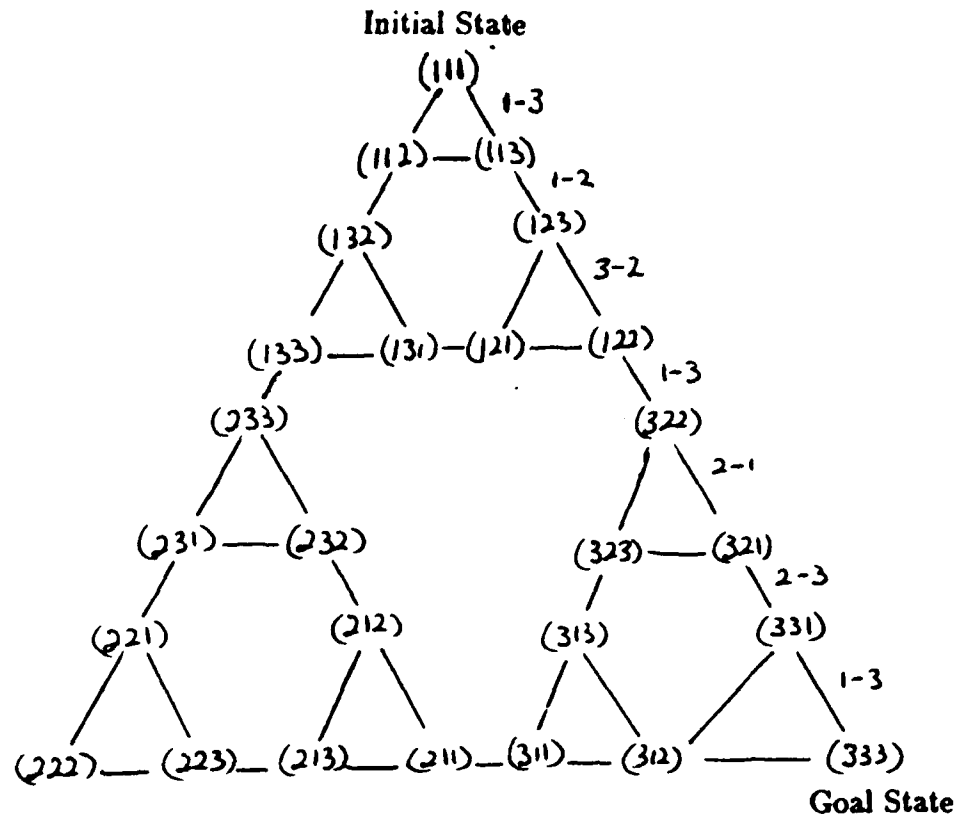


Figure 26: State Transition Graph for Three-Disk Tower of Hanoi Puzzle

(Adapted from Nilsson [38])

values that indicate the states' closeness to the goal node. For the state representation described above, this mapping cannot be formed by a linear threshold unit, or other unit based on a weighted sum of its inputs. For the experiments described in this section the representation was simplified somewhat, as described later, to reduce the time required to learn the solution. The fact that new features are still required is shown by the inability of a one-layer system to learn the minimal solution path.

The formation of useful search heuristics for the Tower of Hanoi puzzle is less complicated. A small set of rather simple heuristics can constrain the search to exactly the correct actions [5,31]. For example, many alternatives are removed by a rule stating that it is undesirable to apply the inverse of an action. The action network used to learn search heuristics in our experiments is single-layered, and did successfully learn the minimal solution path. However, it did not do this by learning the simple set of heuristics. We discuss this point later when we compare the network method to Langley's production system [31].

Representation of States and Actions

As mentioned above, the state representation consisting of three peg numbers corresponding to each disk results in a very complex mapping from states to evaluations. Although in principle the networks used here should be able to learn this mapping, we wished to simplify the task somewhat to reduce the simulation time required for the experiments. The state representation used in the following experiments is composed of nine binary digits. The first three digits encode Disk C's peg number, the second encode Disk B's peg, and the third set of three digits encode Disk A's peg. Peg 1 is encoded as 100, Peg 2 as 010, and Peg 3 as 001. For example, state (111) is represented as (100 100 100), and state (123) is represented as (100 010 001). We also use a constant input of value 0.5 so the threshold can be varied.

Specifically, the input terms, $x_i[t]$, for the state at time t are given by:

$$x_0[t] = 0.5,$$

$$x_1[t] = \begin{cases} 1, & \text{if Disk C is on Peg 1 at time } t; \\ 0, & \text{otherwise,} \end{cases}$$

$$x_2[t] = \begin{cases} 1, & \text{if Disk C is on Peg 2 at time } t; \\ 0, & \text{otherwise,} \end{cases}$$

$$x_3[t] = \begin{cases} 1, & \text{if Disk C is on Peg 3 at time } t; \\ 0, & \text{otherwise,} \end{cases}$$

and similarly for $x_4[t]$, $x_5[t]$, $x_6[t]$ and Disk B, and for $x_7[t]$, $x_8[t]$, $x_9[t]$ and Disk A. After the goal state is reached and at the start of every run, the state is set to (111), so the input becomes (100 100 100) disregarding the constant input.

Both the evaluation network and the action network receive the representation of the state. This completely defines the input to the evaluation network, but additional terms are presented to the action network. We wished to investigate the ability of the network to learn search heuristics similar to the rules developed by Langley's SAGE system. As mentioned above, one such rule is to never apply the inverse of the previous action. In order to learn such an association between the previous action and the current action, the previous action must be provided as input to the action network. Another rule learned by SAGE is to not apply an action that returns the puzzle to a state that was visited two steps ago. This avoids the three-step loops around the smallest triangles in the state transition graph (Fig. 26). Rather than providing past states as input, we chose to present the action taken two-steps ago, in addition to the previous action. The previous two actions along with the current state provide enough information to identify the state visited two steps ago, although our results suggest that hidden units are needed to overcome the linearity of the output unit, perhaps by forming a conjunction of the previous two actions. This possibility was not investigated.

The output of the action network represents an action by a six-component, standard unit basis vector, where the components correspond to actions 1-2, 1-3, 2-1, 2-3, 3-1, and 3-2, respectively; Action 1-2 is encoded as (100000), Action 1-3 is (010000), and so on.

Letting the action at time t be denoted by $(a_1[t], a_2[t], \dots, a_6[t])$, the input terms that the action network receives in addition to those also received by the evaluation network are:

$$\begin{aligned} x_{10}[t] &= a_6[t-2], \\ &\vdots \\ x_{15}[t] &= a_1[t-2], \end{aligned}$$

and

$$\begin{aligned} x_{16}[t] &= a_6[t-1], \\ &\vdots \\ x_{21}[t] &= a_1[t-1]. \end{aligned}$$

Reinforcement

The most significant reinforcement occurs whenever the goal state is entered. A reinforcement value, labeled $r[t]$, of 1 is presented for the time step at which the goal state (333) is entered. Recall that for the pole-balancing task, the goal is to avoid certain states for as long as possible, and $r[t]$ was set to -1 upon entering those states. The Tower of Hanoi task could be solved (by a two-layer network) with only this final reinforcement, but two additional reinforcements are provided for the following reasons. If the action probabilities converge too quickly, due to a large value for the parameter ρ , a solution path of longer than minimum length will probably be learned. For example, say the learned solution path is of length eight, one step longer than the minimum number, due to the incorrect Action 1-2 being taken from the starting state (111). If this action is always chosen over the correct Action 1-3, then an evaluation function tailored to this particular solution path will be learned. To avoid this, a second reinforcement signal is presented having a constant value of -0.1 for all non-goal states. In this way, a shorter solution path results in a higher total reinforcement than does a longer solution path. This parallels the role of Langley's heuristic for judging shorter paths between two states as more desirable.

The third reinforcement is a value of -1 presented whenever a two-step loop occurs, i.e., when the current action reverses the effect of the previous action. The

random search initially followed by the action network results in many two-step loops (and longer loops), thus many steps elapse before the goal is discovered. The large negative reinforcement results in a significant decrease in the probability of selecting the action that is the inverse of the previous action. As shown later, this must be learned for each action—the concept of inverse actions is not known to the system, so generalizing across actions is not possible. This reinforcement is not presented to the evaluation network, enabling a large negative reinforcement to be used without decreasing the evaluation for the corresponding state. The large negative reinforcement is not meant to indicate that a state is bad, only that the action was bad. The selection of the inverse action should be discouraged, but not necessarily the visitation of the state.

This third type of reinforcement is not necessary for the system to learn the puzzle's solution. It was included for two reasons. First, it does significantly reduce the number of search steps during the early stages of learning. Second, it demonstrates how domain knowledge, such as the undesirability of one-step loops, can be added by altering the reinforcement function.

The value of $r[t]$ includes just the first two kinds of reinforcement, while the one-step loop penalty is given by $r_{\text{loop}}[t]$ to distinguish between the reinforcements that are and are not presented to the evaluation network:

$$\begin{aligned} r[t] &= \begin{cases} 1.0, & \text{if state at time } t \text{ is } (333); \\ -0.1, & \text{otherwise.} \end{cases} \\ r_{\text{loop}}[t] &= \begin{cases} -1.0, & \text{if state at } t \text{ equals state at } t - 2; \\ 0.0, & \text{otherwise.} \end{cases} \end{aligned}$$

The learning methods are very similar to those used for the pole-balancing task, with small modifications to the network structure and the learning rules. The only modification to the learning rules used for the pole-balancing task involves the equation for updating the weights of the action network. The reinforcement signal for one-step loops, r_{loop} , is added as follows:

$$w_{i,j}[t] = w_{i,j}[t-1] + \rho (r_{\text{loop}}[t] + \hat{r}[t]) (a_j[t-1] - E\{a_j[t-1]|w;x\}) x_i[t-1].$$

Results of One-Layer Experiments

As was done for the pole-balancing experiments, the performance of a system with a one-layer evaluation network was compared to the performance of a system with a two-layer evaluation network. The systems with the one and two-layer evaluation networks are shown in Figs. 28 and 31 respectively. As in the pole-balancing experiments, two performance measures were used to select the best values for the parameters of the weight-update equations. A measure of cumulative performance throughout a run is provided by the number of trials (achievements of the goal state) averaged over all runs for a given set of parameter values. The second performance measure is the average over all runs of the number of steps in the last trial, or the preceding trial, whichever is smaller.

The final performance level averaged over 5 runs of 50,000 steps each was used to select the best of approximately 20 sets of parameter values, differing in ρ and β , leaving $\gamma = 0.9$. The best of these values are:

$$\begin{aligned}\beta &= 0.100, \\ \rho &= 0.01, \\ \gamma &= 0.9.\end{aligned}$$

These values were used in a longer experiment of 10 runs of 100,000 steps each, resulting in the number of trials and last trial lengths shown in Table 16. The

Table 16: Results of One-Layer System

Run	Trials	Last Trial
1	2,911	28
2	2,877	23
3	2,884	19
4	2,893	65
5	2,686	651
6	2,928	25
7	4,481	9
8	2,902	25
9	3,951	38
10	2,940	41

average number of trials is 3,145. From the lengths of the last trial for each run we see that the minimal solution path was not learned in any run—all trials are longer than seven steps. Run 7 resulted in a last trial of length nine, but it wasn't determined whether the path taken on the final trial would be reliably followed for subsequent trials. The low number of total trials for Run 7 indicates that paths longer than nine steps are likely.

The trial length versus the number of trials is plotted in Fig. 27, showing how the length of the solution path varies with experience. The horizontal dotted line in the figure is at a trial length of seven, the length of the minimal solution path. The values plotted are averages over the 10 runs and over bins of 100 trials. The length of the

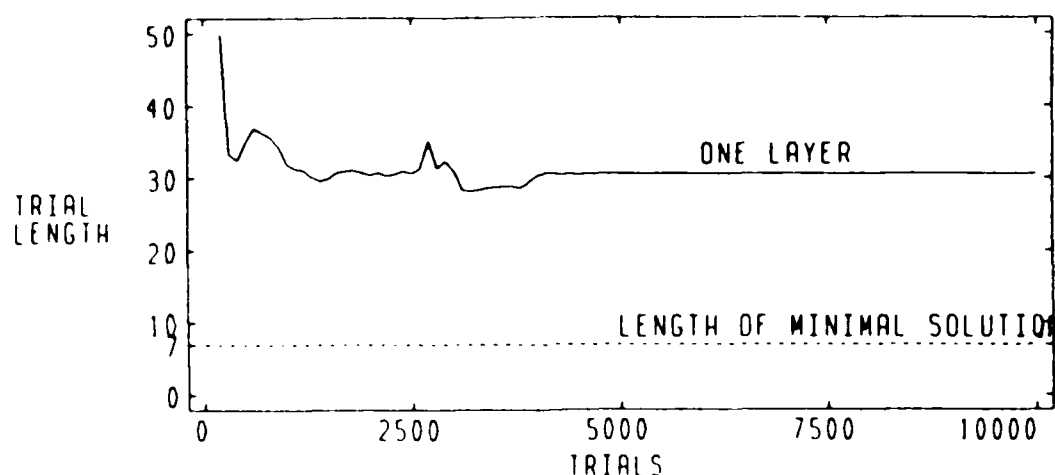


Figure 27: Length of Solution Path versus Trials for One-Layer System

solution path decreases quickly from an average of 50 steps to approximately 30 steps, but performance is never much better than 30 steps per solution. A non-learning strategy of random action selection was found to result in an average of 140 steps per solution path, so the one-layer system significantly improves the initial random search strategy. Note that all runs were terminated before 5,000 trials elapsed—the learning curve was extended as was done for the pole-balancing experiments. The curve might have continued to decrease slightly if the one-layer experiments had been run longer.

The weights learned by the end of Run 7 are shown in Fig. 28. The evaluation network has acquired only three weights of significant magnitude, and they are all

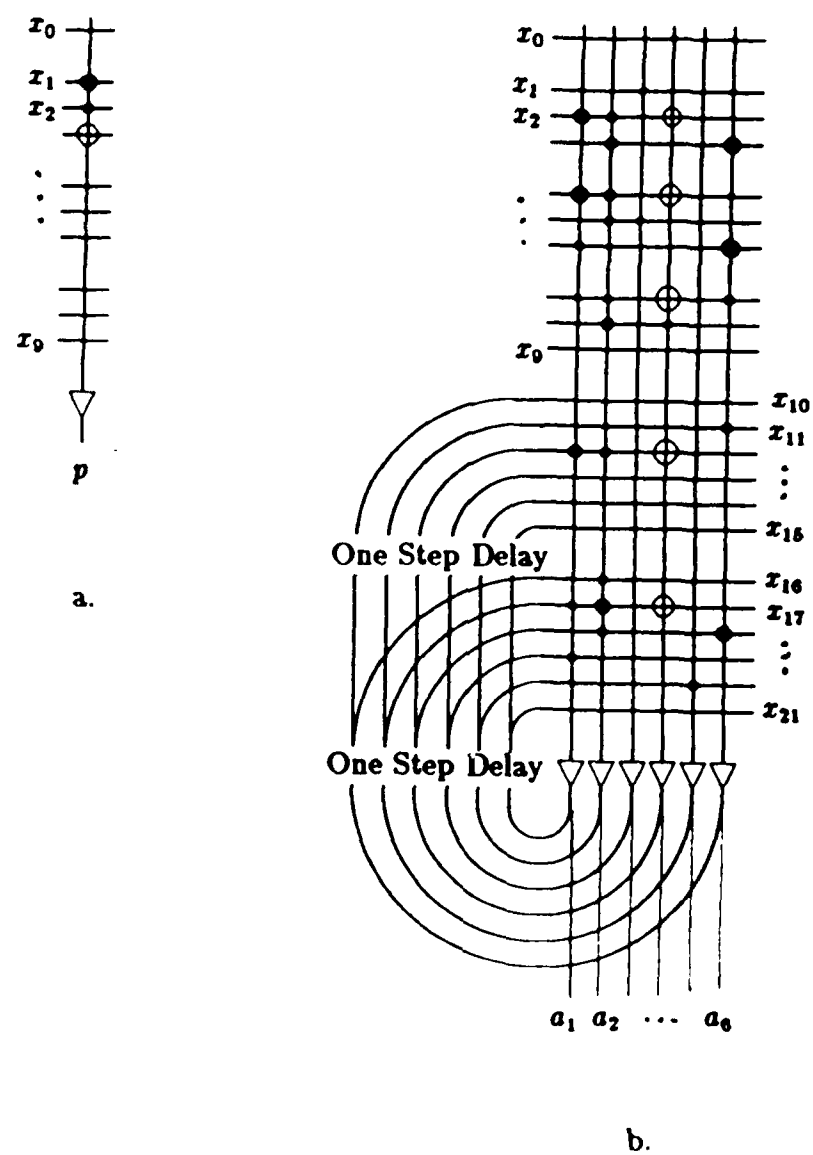


Figure 28: Weights Learned by One-Layer Network

associated with the encoding of Disk C's peg. The weights' signs result in high evaluations for states with Disk C on Peg 3, the goal peg, and low evaluations for other states. The weights of the action network are more difficult to interpret. Let us postpone the discussion of these weights until the results of the subsequent experiments with a two-layer evaluation network are presented.

We can visualize the learned evaluation function by drawing at each node in the state graph a circle with radius proportional to the state's evaluation. The evaluation function learned in Run 7 is shown in this manner in Fig. 29. As determined from the

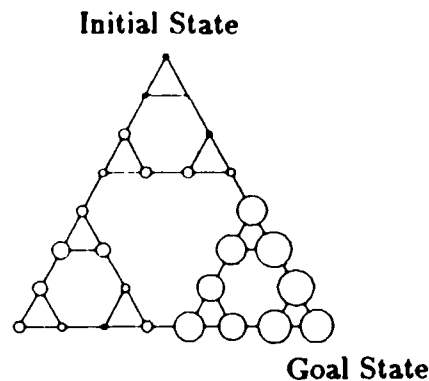


Figure 29: Evaluation Function Learned by One-Layer Network

signs of the weights, the evaluation function indeed produces high values for states for which Disk C is on Peg 3, which are the states in the large, lower right triangle of the state transition graph. There is very little additional information provided by this evaluation function. We can describe this function as a credit-assignment heuristic, viz., states with Disk C on Peg 3 are desirable.

Results of Two-Layer Experiments

Our two-layer experiments involved a two-layer evaluation network with 10 hidden units, but with the same one-layer action network as above. We suspected that with the delayed actions as input terms, the one-layer action network could find weight values that result in the minimal solution path. This is verified by the results of the experiments.

Approximately 20 sets of parameter values were tested by performing 5 runs of 50,000 steps each. The values resulting in the best performance are:

$$\begin{aligned}\beta &= 0.1 \\ \beta_h &= 2.0 \\ \beta_m &= 0.9 \\ \rho &= 0.02 \\ \gamma &= 0.9\end{aligned}$$

Momentum was discovered to facilitate learning in this case, though interestingly it retarded learning for the pole-balancing task. Applying these values in 10 runs of 100,000 steps produced the results in Table 17. For all but one run the length

Table 17: Results of Two-Layer System

Run	Trials	Last Trial
1	11,809	7
2	11,418	22
3	11,584	7
4	11,559	7
5	11,967	7
6	12,093	7
7	11,856	7
8	11,636	7
9	12,432	7
10	12,041	7

of the last trial was seven steps, equal to the length of the minimal solution path. The average number of trials is 11,839, roughly 10 steps per trial averaged over the 100,000 steps. So in 9 out of 10 runs the minimal solution path was learned, and judging from the number of trials in Run 2, the minimal solution path was probably reliably followed in that run, also. There is always a nonzero probability of trying an alternative path, which could explain the last trial of Run 2.

The learning curve of Fig. 30 shows that the two-layer system quickly learned solution paths averaging about 15 steps in length, and gradually reduced this to the minimum of seven steps. The learning curve for the one-layer system is superimposed on this graph to highlight the performance increase resulting from the hidden units in the evaluation network.

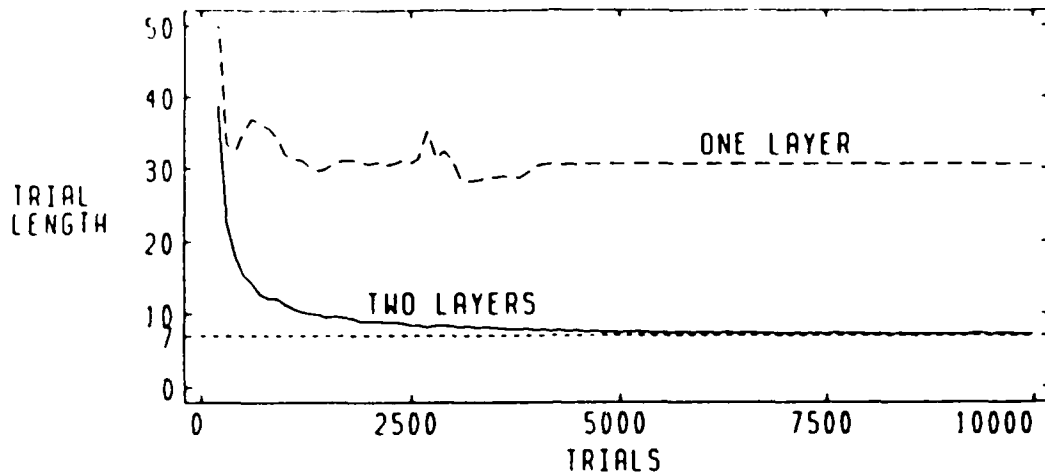


Figure 30: Length of Solution Path versus Trials for Two-Layer System

The weights learned during Run 1 are displayed in Fig. 31. First we focus on the weights of the evaluation network. The hidden-unit weights are more varied than they were for the pole-balancing task. Most of the units appear to have acquired useful new features. Units 1, 2, 5, 6, and 9 have weights of large magnitude, though they are by no means the only units of significance. As is usually the case, it is difficult to comprehend what role the units play by studying individual weight values. However, by encoding their output values by the size of circles on the state transition graph, as done earlier for the evaluation function itself, we can learn exactly what the new features are and can gain some intuitions about their contributions to the overall evaluation function. First, we analyze the evaluation function learned in Run 1.

The value of the evaluation function learned in Run 1 is represented in Fig. 32. In comparing two states, the state with the larger circle would be evaluated as being more desirable. Notice that a consistent progression from small circles to larger circles results as one moves from any state toward the goal state by the shortest route, thus this evaluation function is extremely informative. Any search strategy, in addition to the probabilistic method used to generate actions, would benefit from this evaluation function.

Now let us see how this evaluation function is constructed. Figure 33 shows the output functions for the 10 hidden units, i.e., the features acquired during learning. The radii of the circles for a feature are calculated by scaling the 27 output values for

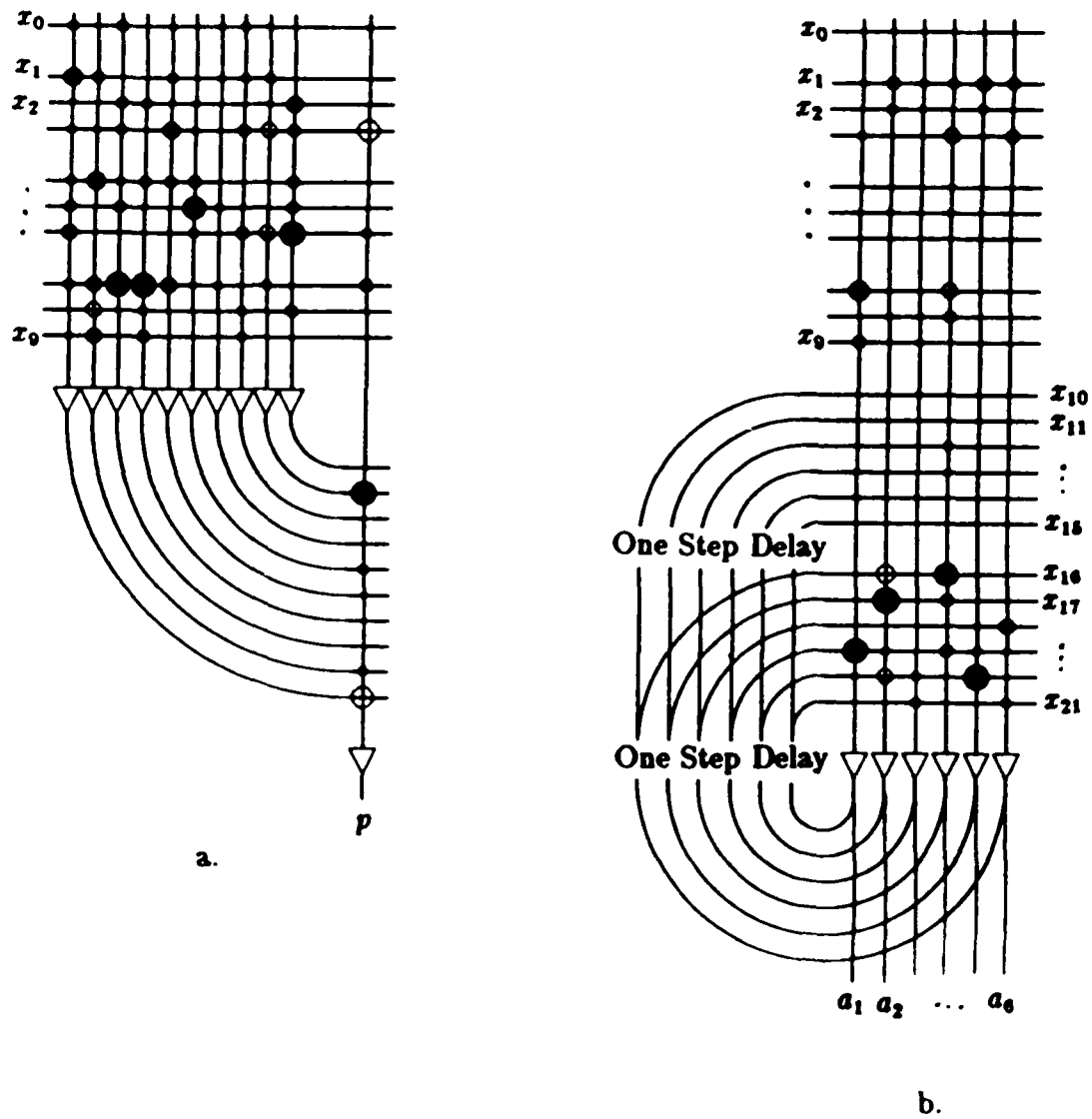


Figure 31: Weights Learned by Two-Layer Network

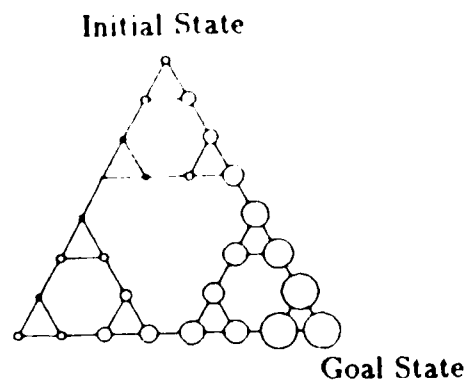


Figure 32: Evaluation Function Learned by Two-Layer Network

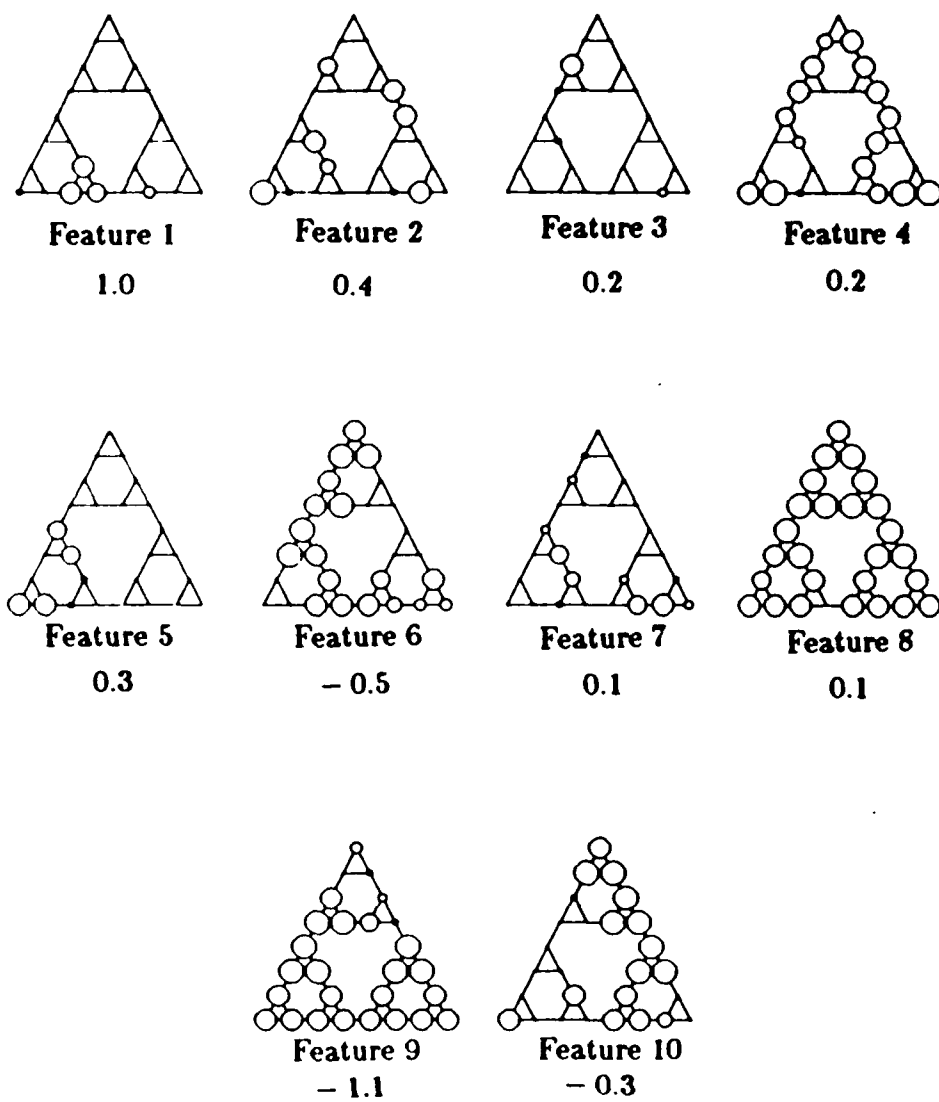


Figure 33: New Features Learned by Two-Layer Evaluation Network

the corresponding hidden unit to be between 0 and a maximum radius. So circles of extreme size do not necessarily indicate that the output is 0 or 1, but only that the output is a minimum or maximum of the values for that unit. We will not attempt to explain every feature, but will consider the contributions of several. We refer to a feature by the corresponding unit number, such as Feature 1 for the function learned by Unit 1.

Feature 1 has a positive effect on the evaluation. Figure 33 shows that Feature 1 roughly represents three states near the bottom of the graph just outside of the lower right triangular region where Disk C is on the goal peg. Feature 1 boosts the evaluation of these states, thus directing a search from states in the lower left part of the graph toward the state through which the search must pass to get Disk 3 onto the goal peg. This part of the graph is a "bottleneck", and similar bottlenecks exist at the other two junctions of the three largest triangles. The values of the evaluation function are critical near these bottlenecks—the choice of an incorrect action can result in many additional moves to return to the bottleneck to try a different action. Features 1 and 2 seem to be particularly helpful in evaluating the lower bottleneck and the bottleneck on the right, respectively.

Feature 9 has a very strong negative influence on the evaluation function. The value of Feature 9 is high for all states except the first four states on the minimal solution path, plus one nearby state. The evaluations of the first states in the solution path are raised in relation to the evaluations of the other states, thus Feature 9's role is to make the first few states of the minimal solution path more desirable than states next to the path.

Feature 10 also has a negative effect. Mainly the evaluations of states along and next to the minimal solution path are lowered by Feature 10, with the exception of the very last state before the goal state. It appears that this feature guarantees that the difference in state evaluations is positive as the last state is reached.

Other features also have important contributions. Perhaps a good way to understand their roles is to observe changes in the evaluation function as each feature is removed and then restored. From the small amount of analysis done here, it is clear that a variety of new features were developed for this task. The initial representation

of the state is far from ideal for forming the evaluation function, but the combination of the error back-propagation algorithm with Sutton's AHC algorithm successfully learned sufficient new features for the state representation.

We now discuss the action function that was learned. When the two-layer evaluation network was used, the single-layer action network was able to constrain the search to the minimal solution path. Figure 31b shows that this was accomplished mainly through the development of weight values for the previous-step's action and for the current state. The two-step delayed action did not acquire a significant effect on the selection of the current action.

The large negative weights on the previous action components stand out. Note that there is one large negative weight for each component. These weights have the same effect as did Langley's heuristic for preventing the application of the inverse of the previous action. By tracing the delayed output of each unit to the corresponding negative weight, we find that the negative weights are on the intersections of actions and their inverses. In other words, Action 1-2, or $a_1[t-1]$, has a negative connection to Action 2-1, or $a_3[t]$, Action 1-3 is negatively connected to 3-1, etc. A negative weight lowers the probability that the corresponding action will be generated, and these weights are of such high magnitudes that the probability of the previous-action's inverse is effectively zero.

There are other weights associated with the previous-action inputs that are positive-valued. Through these weights, the generation of an action on one step results in a high probability for a particular action on the next step, thus forming two-step *sequences*. For example, Action 3-2 will be followed by Action 1-3. Referring back to Fig. 26, this two-step sequence can change the puzzle from the third state on the minimal solution path to the fifth state. Other sequences exist for other two-step transitions along the minimal solution path, and for moving onto the minimal solution path.

The weights on the delayed action components are not sufficient in themselves for limiting actions to movement along the minimal solution path. The current state must at least play a role in selecting the first action. Consider the values of the input terms when in the start state (111). All of the delayed-output terms are 0, since

this is the first step in the trial. All other input terms are 0, except for the first term of each of the three triples encoding the state. The first of these is connected positively to Unit 2 and negatively to the rest, except for Unit 6, whose action is not legal for the start state. The other two non-zero input terms have small or negative connections to units having legal actions, so Unit 2 will be the unit to respond to the start state. Unit 2 represents Action 1-3, the first action along the minimal solution path. Langley's system was not required to learn the correct action for the initial state, because both states (333) and (222) were goal states—two minimal solution paths exist, and both actions from the initial state (111) move along one of the paths.

Transfer of Learning

It is desirable for a learning system to be able to improve its performance on a single task, called *improvement*, and also to improve performance over a set of tasks, called *transfer*. Langley [31] lists the following four kinds of transfer between tasks:

1. Transfer to more complex versions of the task.
2. Transfer to different initial states or goal states.
3. Transfer to tasks of similar complexity with different state-space structures.
4. Transfer to tasks of little similarity, perhaps requiring some of the same actions (referred to as *learning by analogy*).

The ability of the network of this chapter to perform the first two kinds of transfer are discussed below.

Langley showed that the heuristics learned by SAGE for solving the three-disk Tower of Hanoi puzzle were directly applicable to the four-disk and the five-disk versions of this puzzle, solving these more complex puzzles with no additional search. The representation of the rules' conditions and actions made this possible: disk, peg, and action names are generalized to variables, therefore the rules could be applied to the new task having an additional disk, since its name could be bound to a variable. In addition, the concept of an action's inverse is included in the system's

representation, enabling a situation where an action is followed by its inverse to result in a rule discouraging the use of the inverse of *any* action.

The connectionist representation used here does not permit such generalizations, although some learned knowledge is transferred to Tower of Hanoi puzzles having more disks. If the input representation of the networks is augmented by adding components to encode the position of the fourth disk, then some of the resulting action heuristics are wrong and some appropriately transferred. The negative weights preventing the selection of an action that is the inverse of the previous action are still very helpful for the four-disk puzzle. Some of the two-step sequences might also be applicable. To learn the four-disk solution, the evaluation function must also be adjusted, since it is very tailored to the three-disk version. Therefore, the solution of the four-disk puzzle would require additional learning, although probably less than would be needed by a naive system that has no experience with the three-disk puzzle.

The second form of transfer concerns different initial and goal states. Langley's system was not capable of transferring to Tower of Hanoi puzzles with different initial and goal states, but he has shown on another task how the inclusion in the system of a representation of the goal can lead to strategies that are goal-dependent. The action function learned by our system might generalize correctly to different initial states, particularly those close to the minimal solution path, but this was not tested. The evaluation function does generalize correctly to different initial states. As shown in Fig. 32, the evaluations increase for states closer to the goal, whether or not the states are on the minimal solution path. Therefore, learning would be facilitated if the initial state were changed from its original position after the evaluation function had been learned.

As for different goal states, both action and evaluation networks have learned inappropriate functions. In fact, generalization to a puzzle with a different goal state would retard the learning of a new solution path. As Langley suggested, to learn evaluation and action functions for different goals, some representation of the goal must be included as input to the networks. This could be done very simply by duplicating the terms of the current state representation and using them to encode the goal state. Different evaluation and action functions would then be learned for

different goals, though a multilayer action network would probably be required.

Conclusion

In Section 5, a connectionist learning method was applied to a task having a *large search space*, delayed reinforcement, and requiring non-trivial (nonlinear) combinations of features. In this section, essentially the same method was applied to a task with a *small search space*, requiring non-trivial feature combinations, and for which reinforcement is delayed *and* infrequent. We have shown how some of the credit-assignment techniques that have been developed for learning rules *while doing* can be incorporated into a reinforcement scheme.

The adaptive network was able to learn the solution to the three-disk Tower of Hanoi puzzle. The time (amount of experience) required to solve it is much greater than that required by Langley's [31] adaptive production system, but fewer assumptions are incorporated into the design of the connectionist learning method. A very limited input representation is used, consisting only of the current state and the two-previous actions. Comparisons of this connectionist approach with symbolic approaches highlights some of the limitations of connectionist representations. For example, the connectionist system used for the Tower of Hanoi experiments is not capable of doing variable binding in the way that Langley's [31] production system can. Langley's system was able to learn a single symbolic rule that uses action variables to prohibit actions that are the inverses of the previous actions. Langley's production system was able to learn such rules using built-in knowledge of what "inverse" means and how particular actions and states can be generalized to variables. In our implementation, actions are not generalized to variables; distinct negative weights from each action to its inverse had to be learned. Touretzky and Hinton discuss issues of this kind and present some connectionist approaches to these problems [54,53].

SECTION 7

SUMMARY AND CONCLUSIONS

The major focus of the research reported here was the study of layered networks for learning nonlinear associative mappings. We continued our approach to this problem based on the cooperative interaction of self-interested, goal-seeking network components, but we also looked at other approaches. The major results of this research are presented in this report and summarized here in this section. I also include discussion of how the learning methods we have studied relate to existing methods and what avenues appear promising for future research.

The Associative Reward-Penalty Unit

The Associative Reward-Penalty Unit, or A_{R-P} Unit, is a neuron-like adaptive unit that implements a learning rule which is a synthesis of two types of learning methods that have usually been studied separately. Under one set of restrictions, the A_{R-P} learning rule specializes to a stochastic learning automaton algorithm that has been widely studied in the past; under a different set of restrictions, the A_{R-P} rule specializes to a supervised pattern classification method that has also been widely studied (the perceptron learning rule). Consequently, the A_{R-P} rule falls in the intersection of important classes of learning methods. Although the "selective bootstrap learning" rule of Widrow et al. [58] is a very close relative of the A_{R-P} rule, we believe the A_{R-P} rule is novel. The recent pattern classification method of Thathachar and Sastry [51] utilizes stochastic learning automaton methods but is not directed toward solving the same kind of tasks as is the A_{R-P} method.

In Section 2 I discussed what is to be gained by the kind of synthesis represented

by the A_{R-P} rule. Units employing this rule do not need explicit instructional information in order to learn associative mappings and, in fact, are able to learn under extreme uncertainty. This ability has implications for enriching the type of game and team problems that can be studied (as illustrated by our layered-network simulations of Section 3) and for applications to control problems (as illustrated by our pole-balancing and Tower of Hanoi examples of Sections 5 and 6).

Work that remains to be done regarding the theory of single A_{R-P} units concerns their behavior in problems in which the input vectors are not linearly independent. The A_{R-P} convergence theorem applies only to the case of linearly independent input vectors, but the utility of the A_{R-P} rule is not restricted to this case, and it is likely that the convergence results can be extended. The asymptotic behavior of a single A_{R-P} unit needs to be examined in cases in which the input vectors are linearly separable and not linearly separable. Preliminary simulations suggest that A_{R-P} units maximize reward probability in these cases, which is not what the usual methods do. This behavior could have interesting implications that we have not yet explored. We placed higher priority on studying the cooperative behavior of interconnected A_{R-P} units. Although this research direction makes it more difficult to obtain mathematical results, we pursued it because of our basic interest in studying collective behavior.

Cooperative Behavior of A_{R-P} Units

In the same way that the A_{R-P} learning rule can be viewed either in terms of adaptive pattern classification or in terms of stochastic learning automata, networks of A_{R-P} units can be viewed either in terms of connectionist adaptive networks or in terms of the collective behavior of stochastic learning automata. In Section 3, I discussed layered networks of A_{R-P} units from both perspectives. The examples described in that section show how these networks can learn to solve nonlinear discrimination problems. Networks of A_{R-P} units, or layered teams of A_{R-P} units, are therefore examples of systems that can adaptively develop representations in order to form nonlinear associative mappings.

The method for doing this that is most closely related to A_{R-P} networks is the error back-propagation method of Rumelhart, Hinton, and Williams [44], which was presented at about the same time that we first published results of A_{R-P} network simulations. Since then, Williams [61,62] has shown that there is a strong relationship between these methods: They are both gradient following procedures. Whereas gradient information is directly computed via the backward pass in the back-propagation method, in A_{R-P} networks it is estimated via the sampling procedure realized by the stochastic units.

It is therefore not surprising that the back-propagation method is faster than the A_{R-P} method (confirmed by our comparative studies summarized next). So what advantages might the A_{R-P} method have over back-propagation? First, the A_{R-P} method provides a link to a wide body of literature (the learning automata literature) that has not yet been explored by connectionists. I think that a number of interesting consequences may arise from this connection. Second, the A_{R-P} method does not require the complex back-propagation computation. Consequently, it may have some advantages for implementation by parallel hardware and might be more plausible than back-propagation from a biological perspective. Third, the A_{R-P} method might be extensible to the case of recurrent networks with asymmetric connection matrices in ways that back-propagation is not (some recent results by Williams [61,62] are relevant in this regard). Additional research is needed to explore these possibilities.

One of the most important questions regarding network learning methods is how well they scale up to larger, more difficult problems. The research covered by this report does not address this question. What we have learned about the A_{R-P} network method, however, suggests that a straightforward scaling up of the method will not be effective. By a straightforward scaling up of the method I mean that the single reinforcement signal is just broadcast to a larger number of A_{R-P} units. As in the case of the error back-propagation method, as networks get larger, the number of possible solutions can increase so that learning can occur faster for bigger networks. However, in the A_{R-P} method, as networks get larger, the amount of noise that contaminates the gradient estimates increases, a problem that is not present in the

back-propagation method. Thus, while both methods probably scale poorly, the straightforward scaling up of the A_{R-P} method is likely to be worse.

One can, however, consider ways of scaling up the A_{R-P} method that are more interesting than just adding more units and uniformly broadcasting a single reinforcement signal to all of them. It seems clear that the only way to move toward large, complex learning tasks is to use modular or hierarchical networks with local forms of reinforcement. I envision networks in which superordinate modules learn how to provide different levels of reinforcement to different subordinate modules. This approach will involve game decision problems in addition to the team decision problems discussed in Section 3. Consequently, the ability of units employing stochastic learning automaton principles to learn in game situations may be an important factor in implementing these more sophisticated forms of structural credit assignment. This is an important topic for future research.

Comparison of Methods for Learning by Layered Networks

Eleven hidden-unit learning methods were compared by applying them to the task of learning a multiplexer function. The methods were tested in the hidden units of a two-layer network. Two kinds of performance measures were used: the number of errors accumulated throughout a training run and the total number of input vectors for which the final weight values of a run result in an incorrect output. Care was taken to try different parameter values for each method and to present performance measures as averages and confidence intervals over repetitive training runs.

The learning method with the best performance of the algorithms compared was the error back-propagation algorithm of Rumelhart, Hinton, and Williams [44]. The next best performing methods were the reinforcement-learning methods based on the A_{R-P} rule. Best among these methods was a modification of the A_{R-P} rule designed to combine reinforcement learning with a method to create features to represent input patterns present when the network is receiving low reinforcement. A less successful modification of the A_{R-P} method is based on the idea of providing each hidden unit with a more informative evaluation signal than is provided by a reinforcement signal

broadcast to all the units. In this method, reinforcement values are back-propagated based on the weight with which the hidden units are connected to the output unit (unlike the back-propagation method of Rumelhart et al., reinforcement is back-propagated instead of error). This modification produces faster error reduction early in learning runs, but later in runs the rate of error reduction slows and is surpassed by that of the unmodified A_{R-P} method.

Some more conventional optimization techniques were also applied to the problem of finding weight values for the multiplexer task. These methods perform a direct search in the space of all possible value assignments to the weights of the hidden units. They do not use any knowledge of the network's structure. Such a large search space and the ignorance of the network's structure results in very poor learning performance compared to the other methods tested. We included these methods primarily to serve as control simulations. One of these methods, for example, is probably the simplest possible search technique. To be of any interest at all, a method must perform better than this method.

We also experimented with the idea of improving the accuracy of the gradient estimate produced by the A_{R-P} method by letting each hidden unit try several actions while each of the training vectors is present. We call this method the batched A_{R-P} method. The results of these simulations show that it is possible to obtain increasingly accurate gradient estimates without requiring a complex error propagation process. Letting the hidden units obtain 10 samples for each training vector presentation, we obtained learning in the multiplexer task several times faster than the unbatched method (1 sample per presentation) in terms of the number of presentations of each training vector. Of course, the amount of processing required for each presentation is greater than in the unbatched method, and the actual time required will depend critically on how the system is implemented. If in some learning domain it is costly to obtain stimulus vectors, but it is not costly to update the network and obtain evaluations, then it might be practical to use the batched method to increase the speed of learning.

As in all empirical studies, it is important to stress that the results presented in Section 4 are valid only for the particular task and training regime that was used

during the experiments. For example, a task requiring a smaller network might be most readily solved by a random search of the entire weight space. In fact, in selecting a task for the comparative study, a small task with two input components was tested and it was discovered that a random search solved the task faster than the error back-propagation and reinforcement-learning algorithms. The multiplexer task was chosen because the weight space is sufficiently large that direct optimization methods are slow but not so large that a prohibitive amount of simulation time would be needed to gather significant performance statistics. Time also prohibited the extension of the comparative study to other, more complex tasks as would be required to address issues regarding how well the algorithms scale up to harder tasks.

Strategy Learning with Multilayer Networks

Strategy learning can be characterized as the acquisition of a method for generating actions that cause desired transitions among the states of a problem. The desirability of particular transitions is often indicated by an evaluation that imposes a preference ordering on the possible transitions from a given state. In previous research, we have shown that reinforcement-learning methods can be used to learn to select the best action under these conditions, whereas most connectionist learning methods require knowledge of the correct action.

For some tasks, an evaluation is not immediately available but occurs only after a sequence of actions has been generated. Sutton [47,46] has developed the AHC learning rule for dealing with this temporal credit-assignment problem. Hampson [22] has developed a similar method. Barto, Sutton, and Anderson [13,47,15] combined the AHC rule with a reinforcement-learning method into a single-layer network for strategy learning. In the research reported here, we extended these learning methods for single-layer networks to methods for learning strategies with multilayer networks.

We chose the error back-propagation method to update the weights of the hidden units in networks because it was shown to be fastest by our comparative simulations. Consequently, the strategy-learning networks we studied consisted of the following. For each task, there is an evaluation network and an action network. The evalua-

tion network consists of an AHC unit as output unit and a layer of hidden units. The AHC's error term is back-propagated to the hidden units in exactly the same manner that the error term of the delta rule is back-propagated in the Rumelhart et al. method. The action network consists of an associative reinforcement learning unit as output unit and a layer of hidden units. This unit's "error term" is back-propagated to the hidden units. This hybrid system was applied to the pole-balancing and Tower of Hanoi tasks.

Pole Balancing

The major difficulty in the pole-balancing task is due to the use of a very uninformative evaluation signal. Task-specific information, such as the dynamics of the pole, were assumed to be unavailable to the design of the learning system. The evaluation signal was supplied only when the pole fell or the cart hit the end of the track. Other information concerning the task objective, such as the advantage of maintaining the pole near the vertical, was not assumed. Of course, if such information is available it should be incorporated into the initial design of the learning system or used to provide a more information evaluation or error signal. Our interests, however, are in developing learning methods for those parts of a task for which a minimum amount of information is available. For example, we were not attempting to design a learning control method for the pole-balancing task *per se*—much more information is available for this task than we were willing to use.

The combination of the error back-propagation method with the AHC and reinforcement-learning methods was successful: the two-layer system balanced the pole for many more steps than did a one-layer system receiving the same representation of the pole's state. The hidden units learned features with which the output units could overcome the limitations imposed by the representation of the pole's state and the linearity of the output functions. In analyzing the new features that were formed, it was discovered that only a small number of new features were needed to solve the task. Some runs resulted in the formation of a single new feature, while others resulted in up to three features that developed significant influence on the system's output.

Our previous experiments [13,47,15] with the pole-balancing task involved a single-layer network for which the continuous state space was discretized into 162 distinct, 4-dimensional rectangles to allow the system's units to learn appropriate functions. The networks whose study is reported here differ in the absence of this "decoder" and the addition of hidden units that learn features that decode the state into an appropriate form. Another difference, which makes performance comparisons with our earlier pole-balancing studies difficult, is that after every failure the state of the pole is set to a random state instead of the zero state (vertical, stationary pole), as was done in the previous experiments. For this reason, many more failures were generated in the current paradigm, because some reset states were very near failure states. After the same number of training steps, the current system had not attained as high an average balancing time as had the previous system. This is due to either a) the additional experience needed to learn useful new features, or b) the lack of experience in critical states (such as the zero state) for which nearby states require opposite actions.

Tower of Hanoi

The learning methods used in the pole-balancing network were applied with few modifications to the Tower of Hanoi puzzle. Similar restrictions on the amount of a priori knowledge were assumed. A final reinforcement at the end of a successful sequence of actions (as opposed to an unsuccessful sequence for the pole-balancing task) provided information regarding the objective of the task. The state of the puzzle was presented to the network as a binary vector representing the peg on which each disk resides. The two-layer network again performed better than a single-layer network. The two-layer network reliably found a minimum-length solution, i.e., the network applied a sequence of actions consisting of the minimum number of actions required to achieve the goal state. In solving the puzzle, an evaluation function was learned that ranked states according to the smallest number of moves between the state and the goal state.

In learning the evaluation function, a number of new features were developed by the network. In the state-transition graph for the Towers of Hanoi puzzle there are

several bottlenecks—parts of the graph are interconnected by a single path. New features were formed that discriminate states in the bottlenecks from other states. The output unit of the evaluation network could not learn a monotonically-increasing function through the bottlenecks with the original features, but the new features resulted in a good evaluation function.

Langley [31] developed an adaptive production system that learned to solve the Tower of Hanoi puzzle. The connectionist system that we applied to the Tower of Hanoi puzzle has few similarities to Langley's production system. It is instructive to analyze the differences and to question whether or not they represent fundamental distinctions between symbolic and connectionist approaches. One difference is that Langley uses a full history of past states and actions to aid the assignment of credit, whereas the connectionist system relies on the learning of a good evaluation function to solve the credit assignment problem. This difference is not fundamental to the representations involved; evaluation functions can be used for symbolic systems and a history of states and actions can be of use in training a connectionist system. A history could be used much as it is for the symbolic system, by retrieving the events as training instances. A separate issue is the association by a connectionist system of past events with current action probabilities in order to base decisions on previous states and actions. In our experiments, the connectionist system does receive the two previous actions as input, so two and three step sequences can be learned; the inclusion of all past events as input to the system is not feasible. An alternative is to collapse the history into a weighted average of past events.

Another difference is that a breadth-first search is not performed by the connectionist system. In its initial, naive state, the connectionist system chooses actions randomly and as the evaluation function develops, the action probabilities become increasingly biased towards actions that result in state transitions producing positive changes in the evaluation function. Breadth-first search control can be added to the connectionist framework by disregarding the probabilistic generation of actions and presenting state-action pairs as training instances after some process has assigned credit to every pair. Learning an evaluation function in this case requires the extraction of desirable paths from a state history. One attraction of the connectionist

approach demonstrated here is its ability to learn with minimal resources for search control and history maintenance.

A very important distinction that is currently a topic of debate is the use of *variables*. A single symbolic rule can be applied to many situations through the binding of variables. For example, Langley's system learns a rule that, through variable binding, can be used to avoid the application of the previous action's inverse for all possible actions. With one training instance and knowledge of what an action's "inverse" means, a single rule is learned that generalizes to all other actions. It is not clear how knowledge of an action's inverse can be used in a connectionist system to either a) learn the connectionist analog of a generalized rule with variables, or b) duplicate the weight changes due to experience with one action to the weights of other actions. Touretzky and Hinton [54] have shown how variable binding can be performed in a particular connectionist system.

Related to the issue of variables is the issue of the transfer of learning. After learning strategies for solving one task, an efficient learning system must be able to exploit common aspects between this task and subsequent tasks by applying in similar situations the strategies that worked well for the first task. Langley's production rule having a variable action and state can be immediately applied to other, more complex Tower of Hanoi puzzles. This is not possible for the connectionist system and the state and action representations used here. Learning is transferred but not to the degree possible with variablized rules. The strategies learned from the 3-disk Tower of Hanoi are specifically dependent on the 3 disks—the addition of another disk does not affect the strategies until further learning occurs. Different representation of states and actions would result in different amounts of transfer.

Further Developments of Strategy Learning Networks

Our strategy learning networks can be viewed in the context of two theoretical traditions, and future research can take two directions depending on which tradition is followed. One tradition is that of control theory—adaptive and learning control. The learning methods employed by our networks are related to some discussed in the

past by control theorists, e.g., Refs. [20,34,57,60]. These methods differ substantially from the more orthodox adaptive control techniques that involve the identification of unknown plant parameters in that they can be applied with fewer assumptions about the structure of the plant to be controlled. However, these methods do not lend themselves to rigorous convergence results and so have not been actively pursued by modern control theorists.

I think that the methods illustrated by the results reported here contribute several new methods to this "unorthodox" approach to learning control. One contribution is the use of layered networks to learn nonlinear control rules. Second, the AHC method developed by Sutton for dealing with temporal credit-assignment may be a significant novel method. Finally, the A_{R-P} learning rule is applicable to these types of control tasks (although the strategy learning networks discussed in this report do not use it). In order to continue the development of these methods within this framework of learning control it is necessary to develop the theory as much as possible. Although I do not think one will be able to prove broad convergence results for these types of methods applied to nonlinear control problems, I think that the methods need to be developed to the point where they can be applied more routinely. In order to accomplish this, these methods need to be applied to control tasks that are *simpler* than the pole-balancing task studied here so that network design decisions can be made with the aid of relevant theory and results can be compared with those obtainable by more conventional methods. Of course, the eventual goal is to develop learning control methods for problems to which the conventional methods are not applicable.

The other tradition to which our work can be related is the symbolic artificial intelligence tradition illustrated by the adaptive production system of Langley to which we compared the Tower of Hanoi network in Section 6. In order to make closer contact with this tradition it is necessary to develop more sophisticated representational schemes that facilitate the kinds of functions accomplished by variables and variable binding. It also seems necessary to develop a means for networks to perform something like multistep reasoning processes. Efforts in these directions are being made by some connectionist researchers (e.g., [54,53]), but I know of no "nat-

ural" connectionist way of doing these things. I think that a first step toward these types of capabilities is to develop means for networks to adaptively form internal models of their environments which they can manipulate for a variety of purposes. We and others have taken a few steps in this direction (e.g., [49,50,43,25]), but much progress remains to be made. This direction of research is also relevant for potential applications of networks to the types of engineering control problems discussed above.

BIBLIOGRAPHY

- [1] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski. A learning algorithm for Boltzmann machines. *Cognitive Science*, 9:147-169, 1985.
- [2] S. Amarel. *Problems of Representation in Heuristic Problem Solving: Related Issues in the Development of Expert Systems*. Technical Report CBM-TR-118, Laboratory for Computer Science, Rutgers University, New Brunswick, NJ, 1981.
- [3] C. W. Anderson. *Feature Generation and Selection by a Layered Network of Reinforcement Learning Elements: Some Initial Experiments*. Technical Report 82-12, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, 1982.
- [4] C. W. Anderson. *Learning and Problem Solving with Multilayer Connectionist Systems*. PhD thesis, University of Massachusetts, Amherst, MA, 1986.
- [5] Y. Anzai and H. A. Simon. The theory of learning by doing. *Psychological Review*, 86, 1979.
- [6] A. G. Barto. Learning by statistical cooperation of self-interested neuron-like computing elements. *Human Neurobiology*, 4:229-256, 1985.
- [7] A. G. Barto and P. Anandan. Pattern recognizing stochastic learning automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:360-375, 1985.
- [8] A. G. Barto, P. Anandan, and C. W. Anderson. Cooperativity in networks of pattern recognizing stochastic learning automata. In *Proceedings of the Fourth Yale Workshop on Applications of Adaptive Systems Theory*, New Haven, CT, May 1985. An extended version of this paper appeared in *Adaptive and Learning Systems*, K. S. Narendra (ed.), Plenum, 1986.

- [9] A. G. Barto and C. W. Anderson. Structural learning in connectionist systems. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, Irvine, CA, August 1985.
- [10] A. G. Barto, C. W. Anderson, and R. S. Sutton. Synthesis of nonlinear control surfaces by a layered associative search network. *Biological Cybernetics*, 43:175-185, 1982.
- [11] A. G. Barto and R. S. Sutton. *Goal Seeking Components for Adaptive Intelligence: An Initial Assessment*. Technical Report AFWAL-TR-81-1070, Air Force Wright Aeronautical Laboratories/Avionics Laboratory, Wright-Patterson AFB, OH, 1981.
- [12] A. G. Barto and R. S. Sutton. Landmark learning: An illustration of associative search. *Biological Cybernetics*, 42:1-8, 1981.
- [13] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835-846, 1983.
- [14] A. G. Barto, R. S. Sutton, and P. S. Brouwer. Associative search network: A reinforcement learning associative memory. *IEEE Transactions on Systems, Man, and Cybernetics*, 40:201-211, 1981.
- [15] A. G. Barto editor. *Simulation Experiments with Goal-Seeking Adaptive Elements*. Technical Report AFWAL-TR-84-1022, Air Force Wright Aeronautical Laboratories/Avionics Laboratory, Wright-Patterson AFB, OH, 1984.
- [16] R. R. Bush and F. Mosteller. *Stochastic Models for Learning*. Wiley, New York, 1955.
- [17] R. H. Cannon, Jr. *Dynamics of Physical Systems*. McGraw-Hill, Inc., 1967.
- [18] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley, New York, 1973.

- [19] W. K. Estes. Toward a statistical theory of learning. *Psychological Review*, 57:94-107, 1950.
- [20] K. S. Fu. Learning control systems—Review and outlook. *IEEE Transactions on Automatic Control*, 210-221, 1970.
- [21] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, New York, 1981.
- [22] S. Hampson. *A Neural Model of Adaptive Behavior*. PhD thesis, University of California, Irvine, CA, 1983.
- [23] G. E. Hinton and J. A. Anderson, editors. *Parallel Models of Associative Memory*. Erlbaum, Hillsdale, NJ, 1981.
- [24] G. E. Hinton and T. J. Sejnowski. Analyzing cooperative computation. In *Proceedings of the Fifth Annual Conference of the Cognitive Science Society*, Rochester, NY, 1983.
- [25] M. I. Jordan. Personal communication.
- [26] A. H. Klopff and E. Gose. An evolutionary pattern recognition network. *IEEE Transactions on Systems, Man, and Cybernetics*, 15:247-250, 1969.
- [27] S. Lakshmivarahan. ϵ -Optimal Learning Algorithms—Non-absorbing Barrier Type. Technical Report EECS 7901, School of Electrical Engineering and Computer Science, University of Oklahoma, Norman, OK, 1979.
- [28] S. Lakshmivarahan. *Learning Algorithms and Applications*. Springer-Verlag, New York, 1981.
- [29] S. Lakshmivarahan and K. S. Narandra. Learning algorithms for two-person zero-sum stochastic games with incomplete information. *Mathematics of Operations Research*, 6:379-386, 1981.
- [30] S. Lakshmivarahan and K. S. Narandra. Learning algorithms for two-person zero-sum stochastic games with incomplete information: A unified approach. *SIAM Journal of Control and Optimization*, 20:541-552, 1982.

- [31] P. Langley. Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science*, 9:217-260, 1985.
- [32] G. F. Luger. The use of the state space to record the behavioral effects of subproblems and symmetries in the tower of hanoi problem. *Journal of Man-Machine Studies*, 8:421-441, 1976.
- [33] N. J. Mackintosh. *Conditioning and Associative Learning*. Oxford University Press, New York, 1983.
- [34] J. M. Mendel and R. W. McLaren. Reinforcement learning control and pattern recognition systems. In J. M. Mendel and K. S. Fu, editors, *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications*, pages 287-318, Academic Press, New York, 1970.
- [35] D. Michie and R. A. Chambers. BOXES: An experiment in adaptive control. In E. Dale and D. Michie, editors, *Machine Intelligence 2*, pages 137-152, Oliver and Boyd, 1968.
- [36] K. S. Narendra and M. A. L. Thathachar. Learning automata—A survey. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:323-334, 1974.
- [37] K. S. Narendra and R. M. Wheeler. An n -player sequential stochastic game with identical payoffs. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:1154-1158, 1983.
- [38] N. J. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, 1971.
- [39] D. T. Politis and W. H. Licata. Adaptive decoder for an adaptive learning controller. In *Proceedings of SPIE Applications of Artificial Intelligence III*, Orlando, FL, 1986.
- [40] D. L. Reilly, L. N. Cooper, and C. Elbaum. A neural model for category learning. *Biological Cybernetics*, 45:35-41, 1982.

- [41] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527-532, 1952.
- [42] F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, 6411 Chillum Place N.W., Washington, D.C., 1961.
- [43] D. E. Rumelhart. Personal communication.
- [44] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, vol.1: Foundations*, Bradford Books/MIT Press, Cambridge, MA, 1986.
- [45] T. J. Sejnowski and C. R. Rosenberg. *NETtalk: A Parallel Network that Learns to Talk*. Technical Report EECS-8601, Johns Hopkins University, Department of Electrical and Computer Engineering, Baltimore, MD, 1986.
- [46] R. S. Sutton. *Learning to Predict by the Method of Temporal Differences*. Technical Report, GTE-Labs, Waltham, MA, 1987.
- [47] R. S. Sutton. *Temporal Aspects of Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, MA, 1984.
- [48] R. S. Sutton. Two problems with backpropagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Amherst, MA, 1986.
- [49] R. S. Sutton and A. G. Barto. An adaptive network that constructs and uses an internal model of its world. *Cognition and Brain Theory*, 3:217-246, 1981.
- [50] R. S. Sutton and B. Pinette. The learning of world models by connectionist networks. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, Irvine, CA, 1985.
- [51] M. A. L. Thathachar and P. S. Sastry. Learning optimal discriminant functions through a cooperative game of automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 17:73-85, 1987.

- [52] E. L. Thorndike. *Animal Intelligence*. Hafner, Darien, Conn., 1911.
- [53] D. S. Touretzky. BoltzCONS: Reconciling connectionism with the recursive nature of stacks and trees. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, Amherst, MA, 1986.
- [54] D. S. Touretzky and G. E. Hinton. Symbols among the neurons: Details of a connectionist inference architecture. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 1985.
- [55] M. L. Tsetlin. *Automaton Theory and Modeling of Biological Systems*. Academic Press, New York, 1973.
- [56] R. Viswanathan and K. S. Narendra. Games of stochastic automata. *IEEE Transactions on Systems, Man, and Cybernetics*, 4:131-135, 1974.
- [57] M. D. Waltz and K. S. Fu. A heuristic approach to reinforcement learning control systems. *IEEE Transactions on Automatic Control*, 10:390-398, 1965.
- [58] B. Widrow, N. K. Gupta, and S. Maitra. Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 5:455-465, 1973.
- [59] B. Widrow and M. E. Hoff. Adaptive switching circuits. In *1960 WESCON Convention Record Part IV*, pages 96-104, 1960.
- [60] B. Widrow and F. W. Smith. Pattern-recognizing control systems. In *Computer and Information Sciences (COINS) Proceedings*, Spartan, Washington, D.C., 1964.
- [61] R. J. Williams. *Reinforcement Learning in Connectionist Networks: A Mathematical Analysis*. Technical Report ICS Report 8605, Institute for Cognitive Science, University of California at San Diego, La Jolla, CA, 1986.
- [62] R. J. Williams. *Reinforcement-Learning Connectionist Systems*. Technical Report NU-CCS-87-3, College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, MA, 1987.

END

9-87

DTIC